The Inform 7 Handbook

by Jim Aikin

version 1.96b (Dec. 16, 2009)

Table of Contents

Foreword

Somebody once said that if you ask five poets to define poetry, you'll get seven definitions. I'm sure I'm misquoting wildly — the original version may not have been about poetry at all — but the point should be clear. There are undoubtedly more good ways to explain the intricacies of the Inform 7 programming language than there are people who have written tutorials on the subject.

This book makes no claim to being the ultimate guide to Inform 7. On the contrary: A number of topics that experienced Inform authors may want or need to know about simply aren't included in *The Inform 7 Handbook*, or are mentioned only in passing. The purpose of this book is to give first-time authors the kind of information and insight that they'll need in order to start using Inform 7 without getting tangled up in a lot of complexities.

I've tried to organize the material in ways that I hope will help newcomers find useful information quickly. I've attempted, as well, to write in a way that makes few or no assumptions about what the reader already knows. If you read the *Handbook* page by page from start to finish, you'll spot a few repetitions; I felt that some minor redundancy would be better than forcing readers to hunt for information.

Inform 7 ("I7" for short) comes bundled with two long and detailed tutorial/reference books, *Writing with Inform* and *The Inform Recipe Book.* (These books are found under the Documentation tab in the Inform application.) Before getting too far along in the process of writing a game, every Inform author should read the Documentation! Several times, in fact. That's the best way to get familiar with the power of Inform.

Some aspiring authors, however, find the Documentation a bit daunting. At times the Documentation seems to assume that a gentle nudge in the right direction will be all that readers will need. Step-by-step instructions, cross-references, and full discussions of the myriad details that authors may need to have at their fingertips are not always provided. Clearly, there's room for a different approach.

The Inform 7 Handbook grew out of my experiences teaching younger students (ages 10 through 15) to write interactive fiction using Inform 7. When beginning students asked me how to do the kinds of things that beginning authors naturally want to do, I sometimes found that the information they needed was scattered through the Documentation, making it hard to find and hard to put together into a clear mental picture. Figuring out how to do some of the most basic real-world programming chores strictly by reading the Documentation may take a bit of study. I've heard comments about this from adult newcomers as well.

In *The Inform 7 Handbook,* information is organized into chapters by task. You'll find a chapter on making rooms, a chapter on how to create characters, a chapter on how to work with the commands that the player of your game may type, and so on. None of the chapters is intended to tell you absolutely everything about a given topic that you might want or need to know; after reading a section of the *Handbook*, you'll often want to refer back the Documentation. I've included cross-references in many places to show what pages you should consult.

Inform 7 is not the only programming language available for writing interactive fiction. Its main competition (if free software can be said to compete) comes from TADS 3. Version 3 of TADS (The Adventure Development System) is, in some ways, even more sophisticated than Inform 7. But its user interface — the way it presents itself to the author — is as different from I7 as night and day. TADS 3 closely resembles traditional programming languages such as C. (If you don't know what that means, don't worry about it.)

The "natural language" programming interface of I7 makes I7 very attractive to those who have no background in computer programming. Also, at this writing the TADS 3 development environment, Workbench, is a Windows-only application. TADS games can be played on the Mac or on a Linux computer, but the slick development tools are Windows-specific. For classroom use, Inform 7 is a better choice not only because the "natural language" aspects of the system may be easier for the newcomer to understand, but because it makes no difference whether a given student uses a Macintosh, Windows, or Linux computer.

If neither the no-compromise complexity of TADS 3 nor the friendly but fuzzy approach of Inform 7 appeals to you, you can also investigate Inform 6, which is a completely different and much more traditional programming language, although it shares the "Inform" name, or a simpler system such as ADRIFT or ALAN. ALAN is cross-platform (Mac, Windows, Linux); ADRIFT is Windows-only. The full version of ADRIFT is not free, but it's very affordable. All of these systems, and others, are available for download on the Web.

The best source I know of for a comparison of Inform 7 with TADS 3 was written by Eric Eve; it's available on the Brass Lantern site (http://brasslantern.org/writers/iftheory/tads3andi7.html). Some comparisons with other authoring systems are available at www.firthworks.com/roger/cloak/, but at this writing (October 2009) the latter page seems to be almost ten years out of date. Stephen Granade has a few suggestions and comparisons in an essay on Brass Lantern, at www.brasslantern.org/writers/howto/chooselang.html.

Like Inform itself, *The Inform 7 Handbook* is free. I hope you find it useful. If there are areas where you feel it could be expanded or improved, I hope you'll fire off an email (good email addresses are midiguru23@sbcglobal.net and editor@musicwords.net) and let me know what you'd like to see. Updated versions may be released from time to time.

— Jim Aikin
Livermore, California
October 2009

Acknowledgments

Thanks go first and foremost to Graham Nelson for the monumental task of creating, developing, and maintaining Inform. Emily Short is tireless in her assistance and support for Inform authors. David Kinder maintains the Windows IDE (Integrated Development Environment) software for Inform 7, and Andrew Hunter the MacOS IDE. Andrew Plotkin's Glulx system allows large games and games with extra features to be written in Inform; without Glulx, Inform would be a much less attractive proposition for game designers.

Like Inform itself, *The Inform 7 Handbook* is, to a great extent, a community effort. Early drafts were read by Michael Callaghan, Eric Eve, Ron Newcomb, Emily Short, Mike Tarbert, and Michael Neal Tenuis, all of whom suggested valuable additions and clarifications. I also took advantage of materials developed for other classes by Mark Engelberg and Jeff Nyman. During the writing process, the experts on rec.arts.int-fiction freely shared their knowledge with me; you'll find their names scattered throughout. I also cribbed a few questions posted on rec.arts.int-fiction and the forum at intfiction.org by novice programmers, and made free use of questions and concerns raised by the students in my own IF classes.

The Inform 7 Handbook was written using OpenOffice Writer, which also handled the very nice cross-referencing and clickable table of contents in the exported PDF.

The images used in this file (other than those taken from the Inform IDE) were harvested in a frenzy of lawbreaking from various websites; it would be difficult at this late date to figure out who to credit for any of them. A number of them were downloaded from a very nice free image site called stock xchng (www.sxc.hu), but by now I'm not entirely sure which ones. Copyright owners who object to this not-for-profit use of their work should contact me; I'll be happy to remove them from the next edition of the *Handbook.*

Thanks to all!

About the Author

Jim Aikin has released several interactive fiction games, including "Not Just an Ordinary Ballerina" (written in Inform 6), "Lydia's Heart" (in TADS 3), "April in Paris" (TADS 3), "Mrs. Pepper's Nasty Secret" (co-written with Eric Eve in TADS 3), and "A Flustered Duck" (in Inform 7). He is the author of two science fiction novels, *Walk the Moons Road* (Del Rey, 1985) and *The Wall at the Edge of the World* (Ace, 1992). His short fiction has appeared in *Fantasy & Science Fiction, Asimov's Science Fiction,* and other magazines. His nonfiction books include *A Player's Guide to Chords & Harmony* and *Power Tools for Synthesizer Programming* (both published by Hal Leonard), and he has written innumerable features on music technology for *Keyboard, Electronic Musician,* and other leading magazines. His personal website is www.musicwords.net.

Chapter 1: Getting Started

So … you'd like to try writing interactive fiction. It looks like fun, but you're not sure where to start. Or maybe you've already started, and now you're getting confused, or you're not sure how to get the results you want.

This book will help you make sense of it all. Every chapter (except the last one, which is sort of a grab bag) is about one specific part of the writing process. You don't need to read the book straight through from start to finish. Feel free to jump around, dipping into whatever chapter looks as if it will have the information you need. If you want to write a story with lots of characters, for instance, you may want to head straight to Chapter 5. If you find that you're having trouble writing things in a way that Inform understands, you can consult the tips on phrasing and punctuation in Chapter 9. Some of the discussion in that chapter is pretty deep, though, so it may not be the best place to start learning. (That's why it's Chapter 9, not Chapter 1.)

Even right here in Chapter 1, there are things you won't need to think about yet when you're just starting out. If you hit something that doesn't make sense, feel free to skip it and come back to it later.

Whether you read this book or some other manual, or just dive in and start trying things without bothering to read a manual at all, you'll soon learn that writing interactive fiction ("IF" for short) is more than just creative writing. Creativity and storytelling are definitely part of the process, but you'll also be doing a type of *computer programming*. "Programming" means you're giving the computer instructions about what to do. And computers are incredibly picky about understanding your instructions! As you start writing IF, you'll find that a single misspelled word or a missing period at the end of a sentence can stop the computer dead in its tracks. Later in this chapter I'll give you a short list of mistakes that I've often seen students make.

This book is about how to use the popular Inform 7 programming language ("I7" for short). There are several other languages in which you can write interactive fiction, as mentioned in the Foreword. If you're using one of them, you'll find a few general tips in this book that you can use. In particular, the discussion of puzzles in Chapter 6 is largely platform-agnostic. Nonetheless, the book is mostly about how to use I7.

I feel Inform 7 is an especially good choice for those who are new to writing IF and also new to computer programming. I7 is designed to make it as easy as possible to get started. At the same time, I7 includes many "power user" features, so it won't limit you in any way if you want your story to include complex things.

Inform 7 is the successor to Inform 6, which is also a popular language for writing IF. The two are completely different, even though they were created by the same person, Graham Nelson, and are both called "Inform." In this book, we'll most often refer to "Inform" and not bother to add the 7, but all of these references are to Inform 7, not to Inform 6.

It's important to note that Inform 7 is, at this writing (October 2009), still being developed. People have been using it successfully to write interactive fiction for more than three years, but it isn't officially finished. The current version is 5Z71. If certain of the details in this book don't seem correct when you try them out, it's possible that you're using an earlier or later version of Inform, in which the features are different. The basic I7 language is pretty well developed, and probably won't change much in the future. But new features will most likely be added, and there may be changes in some of the details.

In addition, the numbering of pages and Examples in Inform's built-in Documentation is quite likely to change. Please note that all of the page and Example numbers given in the *Handbook* refer to version 5Z71.

Downloading & Installing

To get the most current version of Inform, go to the Inform website and navigate to the Downloads page (http://inform7.com/download/). Click on the icon that shows your computer operating system (Macintosh, Windows, or some variety of Linux) and download the program. You can read instructions on how to install it on each of the operating systems on the same page.

As you work with Inform, you'll be creating *projects.* Each game that you create is a separate project. It's a good idea to create a folder called Inform Projects inside your Documents (Windows XP: My Documents) folder. This may be done for you automatically when you install Inform. Each time you create a new project, save it to that folder. Once you've saved your first project to that folder, Inform will remember the location for you.

Is It a Game, or Is It a Story?

In *The Inform 7 Handbook*, we'll refer to an Inform project as either a "story" or a "game," without making much of a distinction between the two words. There are some real differences, though: Some interactive fiction has a strong story and very little in the way of game-type fun. Other interactive fiction is mainly a game, and the story is so weak it might as well not exist at all.

It's up to you to decide how to combine story elements with game elements. For the purposes of this book, the game is a story and the story is a game.

The Inform 7 Program

The first time you launch the Inform 7 program, a dialog box will ask you whether you want to open an existing project or start a new one. Since you don't yet have a project to open, select "Start a new project...". This will open a box in which you can name your project and tell Inform where you want to save it. (Use the Inform Projects folder in your Documents folder.)

When you close this box, the Inform program will open, and you'll see a main window with two pages — a left page and a right page. The left page will be blank except for the name of your new game and the byline (your name as the author). On the right page you'll see the Table of Contents for the Inform Documentation.

What you're looking at is not Inform itself (though you can think of it that way if you want to). What you're looking at is a program that's sort of a container for Inform. This type of program is called an Integrated Development Environment, or "IDE" for short. The Inform IDE has quite a lot of useful features. Before you start writing your first story, take a quick look around the Inform IDE. Its most important features are discussed in the next few sections of this chapter; others are mentioned in Chapter 10.

Normally, you'll want to keep the text of the story you're writing in the left-hand page. The right-hand page will contain various things at various times — the Documentation, the game itself, error messages, and so on. The IDE will sometimes open new *panels* automatically within the right-hand page. But in fact you can display whatever you'd like in either page. If you're studying two related pages in the Documentation, for instance, you could have one open in the left page of the IDE, and the other open in the right page at the same time. You can even run two instances of the Inform IDE at the same time, by opening additional project files. This way, you can keep the Documentation pages open at all times, or copy what you've written from one project to another.

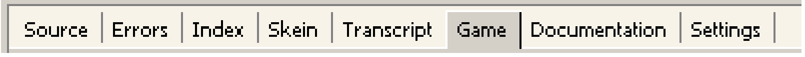
To choose what's displayed in each page, click on an item in the row of page headers. (This is displayed across the top of the pages in the Windows IDE, and vertically along the right edge in the Macintosh IDE.) Each header opens a different panel.

Until you've written something in the Source panel and clicked the Go! button, some of the other panels will be empty. This is normal.

Feel free to click on chapter headings in the Documentation and read a few pages. If you see things that you don't understand (and you will!), please don't worry about it. Even in Chapter 1 of the Documentation, some of the information is fairly advanced, and won't be useful to you yet. Just take a look and get acquainted with how the Documentation is set up.

In *The Inform 7 Handbook,* references to "the Documentation" are generally about *Writing with Inform.* At the bottom of the Table of Contents for *Writing with Inform* you'll see a link to *The Inform Recipe Book.* The *Recipe Book* is an entirely separate manual. Both of them are well worth reading. They share the same set of Examples, which are well worth close study. In fact, some of the pages in these manuals provide only a quick overview of a topic, an overview that is fleshed out with details in the accompanying Examples.

The Page Headers



Across the top of each page, if you're using the Windows IDE, or vertically along the edge if you're using the Macintosh IDE, you'll see a series of panel tabs, or headers: Source, Errors, Index, Skein, Transcript, Game, Documentation, and Settings. Clicking on any of these headers opens up a new panel. We'll save a discussion of the finer points of how to use the Skein, Transcript, and Index for Chapter 10. In this chapter we'll just introduce the panels briefly.

The **Source** panel is where you'll write your game. At the top of the panel you'll see two buttons: Source and Contents. When you start designing your first game, the Contents panel won't be useful; in fact, it will be empty. In large or even medium-sized Inform projects, though, you'll want to add heads and subheads (see "Headings" in Chapter 9) to your source. "Source" is computer jargon for what you're writing. Once you've added some headings, the Contents panel will give you a quick way to navigate around in the source. You can quickly jump from one section to another section that's hundreds of lines away, without having to use the scroll bar beside the panel and hunt for the section you're seeking.

Inform is different from most computer programming languages in that the source for your entire project will be contained in one document (that is, in one file). Using the Contents panel, however, you can navigate through this document in much the same way that you would jump from one source code file to another if you were using a different programming language.

The Source panel includes an important and useful feature called **syntax coloring**. As you start working with Inform, you'll find that different types of text within the source will be displayed in different colors. For instance, text in double-quotes will always be blue. Syntax coloring is used in most forms of computer programming. It's just a friendly feature to make it easier for you to read what you're writing; the colors have no effect on the game itself.

Source: Code or Text?

Computer programmers call what they write *source code.* The term "code" goes back to the early days of computer programming, and probably reflects the fact that most computer programming languages look as abstract and hard to understand as messages encrypted in a secret code. The creators of Inform prefer to refer to what you write as *source text,* because they feel "text" is a friendlier term to use to describe what you'll be writing. However, most of the time, *The Inform 7 Handbook* will refer to what you're writing as "source code" or just "code," because the word "text" in Inform also refers to the sentences you write between double-quotes, which are intended as output during your game. Ambiguity has its uses in literature, and also in real life, but in general it's not desirable in computer programming situations.

The **Errors** panel will open up automatically in the right-hand page when you click the Go! button (see below) if Inform can't figure out what you wrote. Except when dealing with problems, you can ignore the Errors panel.

The **Index** is extremely useful. Until you've compiled your first game, however (see "The Go! Button," below), the Index will be blank. In the process of turning your source text into a game, Inform creates and organizes lists of practically everything in the game. By consulting the pages of the Index, you can learn a lot about Inform, and also spot problems in your game, or just get an energy boost from seeing that your game is getting bigger.

The Index panel is divided into seven pages: Contents, Actions, Kinds, Phrases, Rules, Scenes, and World. We'll have more to say about these in Chapter 10.

The **Skein** is used mainly for testing your game while you write it. The Skein will become a useful tool as your game gets more complex, but when you're just starting out, you can safely ignore it. Briefly, the Skein keeps a record of all of your play sessions as you test your work-in-progress. One of the ways to test your work after making some changes is to open up the Skein, right-click (Mac: control-click) on one of the lozenge-shaped *nodes,* and choose "Play to Here" from the pop-up menu. This will repeat the commands you typed on that particular run-through. For more on the Skein, you can consult **pp. 1.7, 1.8,** and **1.9** ("The Skein," "A short Skein tutorial," and "Summary of the Skein and Transcript") in the Documentation.

The **Transcript** panel is also a tool for advanced programming situations. To be honest, in writing my first large Inform 7 game, I never used the Transcript. To learn a bit more about how to use the Skein and the Transcript, see Chapter 10.

The **Game** panel is where your game will appear, allowing you to try it out as you're developing it. You can't run other games in the Game panel, only the game(s) that you're writing. The appearance of your game in the Game panel will be similar to, but possibly not identical to, the way it will look in an interpreter. Other people will play your game in an interpreter (see "Downloading & Playing Games," later in this chapter), which is a separate program. The text of your game should be identical no matter what interpreter it's played in, but the visual appearance and the type font may not be the same.

As a general rule, relying on the visual appearance of your game to give players important information is a bad idea. Not only will the appearance vary from one interpreter to another, but interactive fiction is popular among blind computer users, because they can play games of this type. In general, anything that your players will need to know to play the game should be presented as ordinary text.

The **Documentation** panel contains two large tutorial/reference works: *Writing with Inform* and *The Inform Recipe Book*. Both of these are hugely valuable resources, and you'll want to get to know them. Also in the Documentation panel is an index of more than 400 Examples showing how to use Inform.

The **Settings** panel is not often needed. At a certain point, though, when your game gets big enough, you'll run into an error message saying that the game can't be built. At this point, you'll need to go to Settings and switch from the Z-code version 5 format to Z-code version 8, or from Z-code version 8 to Glulx. If your game contains images or sounds, you'll need to switch to Glulx no matter how large or small the game may be, and also check the "Create a Blorb archive for release" box before clicking the Release button in the main toolbar. Inform's ability to handle images and sounds is cool, but fairly limited. We won't have much to say about it in this book.

The Go! Button

In my kitchen I have a wonderful invention called a Zojirushi bread machine. It bakes home-made bread. All I have to do is measure the raw ingredients, pour them into the pan, close the lid, and press the Start button. Little paddles inside the machine knead the dough. Then the machine waits an hour for the dough to rise before it heats up and starts baking. Three hours after I put in the ingredients, I have a steaming hot loaf of fresh bread.

Inform's Go! button is a lot like the Start button on my bread machine. The text you write in the Source panel is the raw ingredients — the flour, water, sugar, yeast, and so on. When you click the Go! button, Inform churns and kneads what you've written and turns it into a game that can be played. This function is explained on **p. 1.4** ("The Go! button") of the Documentation.

This process looks very simple — you just click a button. But Inform does a huge amount of work to translate your source text into a game. This work is done by a software "machine" called a **compiler**. There's no need for you to be concerned about how the compiler works its magic. But since we'll be referring to it here and there in this book, you need to know that it exists, and what it does. The compiler turns your source text into a playable game.

If I put too much water into my bread machine, the machine won't know. It will just produce a "loaf" that's a soggy mess. If I forget to add the yeast, the machine will go ahead and do its thing, and I'll end up with a hard, teeth-breaking lump. The bread machine doesn't look at what I put in the pan — it just runs through its process, step by step, automatically.

A compiler is a lot smarter than a bread machine. In order to do its work, the Inform compiler has to scrutinize every single line that you've written and figure out what to do with it. If it understands what you've written, a few seconds after you click Go!, your game will pop up in the right-hand page, ready to play. More than half the time, though, the compiler will encounter problems. It will hit a word, sentence, or paragraph that it can't make sense of. Instead of your game appearing in the right-hand page, you'll see a page in which Inform will report on the problems it ran into.

This is nothing to worry about — it's normal. What you need to do is find the problems, fix them, and click Go! again. Depending on the exact problem(s), you may have to go through this cycle five or six times before your work-in-progress will compile successfully.

Next to each problem report in the right-hand page, you'll see a little orange arrow. When you click on this, the Source page will jump directly to the problem paragraph, which will be highlighted. Now you know what you need to fix.

Each problem message will give you some information about the type of problem the compiler ran into. Sometimes these explanations will make instant sense. Other times, they're more confusing than helpful. If you accidentally type a semicolon (;) where you should have typed a colon (:), for instance, Inform won't tell you that that was the problem. Instead, you'll see a paragraph that reads more or less like this, "The phrase or rule definition ... is written using the 'colon and indentation' syntax for its 'if's, 'repeat's and 'while's, where blocks of phrases grouped together are indented one tab step inward from the 'if ...:' or similar phrase to which they belong. But the tabs here seem to be misaligned, and I can't determine the structure." In this case, Inform will also tell you the specific line where it hit the problem — but the line it identifies is actually the line *after* the typo.

After a while you'll start to get a feel for the types of errors you usually make, and the problem messages you'll see as a result. For more details, see "All About Bugs," later in this chapter. But if all goes well, your game will appear in the right-hand page, in the Game panel, ready for you to try it out.

Five Common Problems

As I watch beginners start to learn Inform, I see certain kinds of problems showing up over and over. In no particular order, here are some things to watch out for in your code:

Forgetting to say "say". You want the game to produce a certain text output in a certain situation, so you just write it, surrounding it with quotation marks as usual. In a couple of cases, such as descriptions of rooms, this is okay. But if you're writing a rule in which you want Inform to say something, you have to say "say" before you start the quoted text.

Missing period at the end of a sentence. Always end sentences with periods. There are some special rules about placing periods in relation to quotation marks, but if the sentence doesn't have a period at all, Inform may get confused. If the sentence is in the middle of a paragraph, Inform is almost certain to get very confused.

Misspelled word. An easy problem to run into, and sometimes hard to spot. You'll be typing the word "description" a lot, and at a high screen resolution you may not notice at first that you've spelled it "descripton" or "descrpition".

Colon instead of semicolon or vice-versa. These two marks look almost alike on the screen, but they're completely different. A semicolon is like a stop sign on the street; it tells Inform, "Okay, stop here for just a second, and then go on, because we're not done yet." A colon is like an arrow pointing forward — it tells Inform, "Do this next."

Wrong indents. See page 233 in Chapter 9 of the *Handbook* for a full explanation of how Inform handles indentation.

One thing you may want to know about the Go! button is that when you click it, the first thing that Inform does is save your Source to a file on disk. This new file will overwrite anything that was in the file before. So if you're trying out various kinds of changes in your game, you may want to create alternate versions of the game — one to experiment with and one as a safe backup copy. To make a separate copy of the game that you can experiment with, use the Save As... command on the File menu.

The Release Button

When your game is finished — or maybe not quite finished, but far enough along that it's ready to share with other people so they can test it, find problems, and offer suggestions — you can click the Release button. This will produce a game file in the .z5, .z8, or Glulx format. You can give this file to other people, or attach it to an email and send it to them. They'll be able to load it into an interpreter and play your game.

Some of the features that you can use while testing your game in the IDE (see "Using the Debugging Commands," later in this chapter) will not be available in the release version.

Other Features

The Stop button is not used often, but you won't be able to make certain kinds of edits in the Skein while the game is running. (That would be like trying to change to a new pair of ice-skates while skating around the rink.) The Stop button will end the current play session, making these Skein edits possible.

Using the Built-In Documentation

Many people use Inform's built-in Documentation as a combined tutorial and reference guide. When starting out, you'll probably want to use it more as a tutorial. To do this, read Chapters 2 through 7 and try out the Examples. The examples can easily be copied into a test game so you can try them out and experiment with them.

First, start a new project or open up a project that you don't mind trashing. I keep a game called Test Game on my hard drive for this purpose. (Actually, I have dozens of them.) You pretty much have to start with a blank Source when trying the Examples, because pasting the Example code into an existing game will most likely make a mess of the game.

Open up the Example by clicking on the blue lozenge-shaped button with the number. Starting with an empty Source page in your Test Game, click on the square button by the first line of the code in the Example. This button will copy the entire Example over to the Source page. Now all you need to do is click Go!, and the Example will turn into a game. After playing the game to see what it does, try changing some of the source code and play it again. This is a great way to learn how Inform works.

Once you've read through large parts of the Documentation (perhaps several times), it will get easier to find the information you need. The Documentation has no index, but the IDE includes a Search field. You can type whatever you'd like in this field, and Inform will go through all of the Documentation looking for matches. You may find anything from no matches to dozens of them. The more specific you can be about what you're looking for, the better the Search engine will work. Phrases like "end the game" and "omit contents in listing" work better than something like "scenery", which produces way too many matches.

At present (in build 5Z71), the Search engine doesn't search the text of the *Recipe Book*, only *Writing with Inform* and the Examples. In the Windows IDE, the search results will be highlighted in the text, making the word or phrase easy to find. The Macintosh IDE does not yet have this neat feature.

Once you've successfully compiled your game, the Index panel will contain numerous links (the blue question-mark buttons) to the Documentation. At a certain point, you'll want to explore the Index, because it has links to lots of good information.

What Happens in a Game

If you've played a few text-based interactive fiction games, you won't need to be told how IF works. But for the benefit of those who may just be getting started, we'll cover the basics here. In a game, you play the part of a character in a story. The story may have a simple concept, or it may be quite complex, involving many characters, locations, and events. In a simple story, you (the character) might be wandering around in a cave, collecting treasures. In a complex story, you'll most likely meet other characters. You may need to outwit them or make friends with them. The story may have several different endings, some of them happy and others not so happy. One of your tasks will be to choose the actions that will lead to a happy ending. But until you start taking actions, you won't know which choices lead to which endings.

Entering Commands

To play the game, you type *commands* (instructions to the computer) when you see the command prompt. The command prompt generally looks like this:

```
>
```

Unlike a video game, where you need quick reflexes to deal with animated characters who are moving around, in a text game nothing happens until you type a command and hit the computer's Enter key. When you press Enter, the game reads your latest command and prints out a response.

Some of the responses you read may be error messages: Maybe you misspelled a word, for instance, so the game doesn't know what you meant. Other responses will show you what's happening in the story. The command you entered may cause a change in the world of the story, or it may just give you more details about what's going on. If you're playing a game, you don't often need to pause to think about how the game reads your commands and responds to them. But if you want to write games, you'll need to know a few details. The software gadget that reads and interprets what the player types is called a **parser**. Every text game has a parser (although the parser used in games that are written using the ADRIFT programming system is so crude as almost not to be worthy of the name). The parser that's built into Inform 7 is very sophisticated. It's able to understand quite a variety of inputs from the player. (You can also change what it does, if you need to. Some of the techniques for doing that are explained in this *Handbook.*)

But no parser is able to understand the kinds of complex sentences that you and I speak and write every day. The parser is designed to process simple commands, such as OPEN THE DOOR and PICK UP

THE BALL. Most commands are of the form <VERB> <NOUN> <MAYBE A PREPOSITION> <MAYBE ANOTHER NOUN>.

Some verbs (such as WAIT, SLEEP, and JUMP) are followed by no nouns at all. Some verbs need one noun — for instance, OPEN THE DOOR and PICK UP THE BALL. A few verbs take two nouns, which are (usually) separated by a preposition. Examples would include PUT THE BOX ON THE TABLE and HIT THE OGRE WITH THE STICK.

When you're entering commands during a game, the word THE can always be left out, and most experienced players never use it. They would just type HIT OGRE WITH STICK. That works perfectly.

If an object has a name that includes an adjective or two, you'll probably be able to use just an adjective to refer to it. If the object is called "the tarnished gold crown" in the game, typing PICK UP GOLD should work (at least, it will work if the author has done a decent job of writing the game). This is not necessarily true in TADS games, but Inform is more relaxed about understanding what the player meant.

It's All About You!

In most interactive fiction, you (the player) play the part of a character called "you." When the game prints out a message, such as "You can't see any such thing," it's talking about the character called "you." The pronoun "you" is grammatically in the second person, so we say these games are written in *second person*. A few games are written in which the character is "I" (first person) or "he" or "she" (third person).

In most games, the action is described as taking place right now. For instance, "You walk up the stairs." Grammatically, this is called *present tense.* A few games are written in past tense ("You walked up the stairs.") Second-person, present-tense storytelling is not used much in novels or short stories, but it's very normal in interactive fiction. It's possible to write a story in any combination of person and tense, but the techniques for doing this are beyond the scope of this *Handbook.*

Once in a while, the parser will ask you for more information. For instance, if you're in a room with a gold crown, a gold ring, and a gold orb, when you type PICK UP GOLD the parser will respond, "Which do you mean, the gold crown, the gold ring, or the gold orb?" At this point, the parser will try to interpret your next input as an answer to the question. If you just type ORB, the parser will understand that you meant PICK UP GOLD ORB, and the game will proceed to do that action.

If you're in a room with another character, you can try giving them a command, like this: BOB, PICK UP THE STICK. Bob may or may not do what you want — at the moment, we're just looking how commands are entered. Here, you type the name of the character, then a comma, then the command.

Most interactive stories take place in what's called a *model world*. This is a place that was created by the author of the story. It exists only within the computer — and in fact, only within the game's interpreter software. It might be an ancient castle, the interior of a space station, or a busy modern city. The model world is always built out of *rooms.* The rooms are the locations where the story takes place. A "room" might be a large open field, or it might be the interior of a phone booth. The word "room" is used in IF authoring to refer to each of the locations in the game, whether or not the location is literally a room.

If you want to see what's in the room with you, you can use the command LOOK (this can be abbreviated L). When you LOOK, you'll read a description of the room and its contents. In general, you can only see what's in the room with you; you can't see into any other rooms. Some games have windows you can look through, but basically windows have to be "faked" using some clever programming tricks.

If there's more than one room in the game, you'll be able to move from room to room. This is usually done by typing commands based on compass directions — for instance, GO NORTH. This type of command is so common that it can be abbreviated. You can type N to go north, NE to go northeast, E to go east, SE to go southeast, and so on. Depending on the type of world where the story takes place, the directions you can travel may also include up (U), down (D), in, and out.

The use of compass directions in IF is artificial but convenient. From time to time authors try to come up with alternatives, but none of these has caught on. In "Blue Lacuna" (a large, complex game written in Inform), Aaron Reed used a neat system in which certain words that are highlighted in the text can be used as movement commands. For example, if the word "beach" appears highlighted, typing BEACH will take you to the beach. This system is available as an Extension for Inform. (For more on how to use Extensions, see p. 35.)

As you travel through the model world, you'll encounter various kinds of objects. Some of them will be portable: You'll be able to pick them up, carry them around, and drop them in other locations. Other objects will be scenery, and can't be moved.

The first thing you'll want to do, when you enter a room, is EXAMINE all of the objects that are mentioned in the room description. Most modern games understand X as an abbreviation for EXAMINE, but a few old-school games don't. Read the room description carefully and then X anything that's mentioned. You may discover important details by doing this.

If an object is portable, you'll be able to pick it up. Let's say the object is a bowling ball. The commands PICK UP BALL, TAKE BALL, and GET BALL all mean the same thing. (Inform programmers call this the *taking* action.) To read a list of the items you're carrying, use the INVENTORY command. This can be abbreviated INV, or simply I.

From time to time you'll find containers. A container may be permanently open, like a basket, or it may be something that you can OPEN and CLOSE, like a suitcase. If a container is open, you can try putting things into it using a command like PUT BOWLING BALL IN THIMBLE. Some of the things that can be opened and closed can also be locked and unlocked.

Most games include puzzles. (For much more on puzzles, see Chapter 6.) Some puzzles are easy, and some are fiendishly difficult. These days, many games have built-in hints that will help you if you don't know how to solve a puzzle, but in other games, it's strictly up to your ingenuity to figure out what to do. To learn how to add hints to an Inform game, see p. 256.

Some games are friendly: You may get stuck for a while before you figure out what you need to do next, but the worst thing that can happen to the character whose role you're playing is wandering around (or getting stuck in a locked room) and not knowing what to do next. Other games are cruel. In a cruel game, if you do the wrong thing, your character can get killed, perhaps in a very nasty way. Fortunately, the UNDO command will usually get you out of trouble.

Not always, though. Some games are so cruel that they won't let you UNDO if your character has died. Because of that, it's a good idea to SAVE your game every so often, especially before trying anything that might be dangerous. If you get killed or lose the game in some other way, you'll be able to RESTORE. The RESTORE command opens a dialog box where you can choose a saved file to reopen. By choosing the most recent saved file, you can revert to a point before you got in trouble. (Wouldn't it be nice if real life was like that?)

Downloading & Playing Games

One of the best ways to learn about game design is to download and play a few games. A good place to start looking for games is the Interactive Fiction Database (http://ifdb.tads.org/). On this site you can read reviews of games, search by author, and click links to download the games themselves.

Other resources for finding games include Baf's Guide (http://www.wurb.com/if/) and an online magazine called SPAG (the Society for the Promotion of Adventure Gaming, http://www.sparkynet.com/spag/). Some games are available from authors' websites, but far more are to be found in the Interactive Fiction Archive (http://www.ifarchive.org/).

To play text games, you'll need both the game file itself and a separate piece of software called an **interpreter**. After downloading and installing an interpreter, you'll load the game into the interpreter to play it. (A few games written in TADS and some other development systems are available as free-standing programs, but usually an interpreter is needed.)

On Macintosh OS X, currently the best interpreter is called Zoom. It's available from http://www.logicalshift.co.uk/unix/zoom/. Zoom can play all Z-code (Inform) and Glulx games, as well as TADS games and Hugo games.

On Windows, you may need a couple of different interpreters, depending on what games you want to play. Games written in Inform and compiled to the Glulx format (their filenames end in .blb or .ulx) can be played with a Glulx interpreter. Games written in Inform and compiled to the .z5 and .z8 format use a Z-code interpreter. Several of the latter are available.

Several new interpreters are in various stages of development. The Parchment interpreter, for instance, plays Z-code games in a standard Web browser.

For more on which interpreter to use, and where to download it, a good site is Stephen Granade's Brass Lantern site (http://brasslantern.org/beginners/tadownload.html).

What's This .z5 and .z8 Stuff All About, Anyway?

In the beginning, there was Zork. Well, no, that wasn't quite the beginning. In the beginning was Adventure. Adventure was the very first text-based computer game. It was freely copied and shared by computer users in the 1970s, and was never a commercial product. But around 1980, some clever people saw that they could make money on text-based games. They started a company called Infocom. The first game that came from Infocom was called Zork.

Zork was enormously successful, and led to a series of sequels, which were also successful. But then, in the mid-1980s, computers became fast enough to display graphics. Games that used graphics were much sexier than text games. As a type of commercial product, text games pretty much died.

By that time, though, there were hundreds of thousands of computer users who owned Zork or one of Infocom's other games. These games were available for many different computer operating systems (and at the time, there were a lot more computer operating systems than there are today). The actual game data was the same no matter what type of computer you owned, but Infocom created *virtual machine* for each operating system. A virtual machine ... well, let's not worry about the technical details. The point is, if you had a virtual machine from Infocom for your computer, you could play any of Infocom's games. All you needed was the data file.

The virtual machine developed by Infocom is what came to be called the Z-machine. (Named after Zork, you see.) Even after Infocom closed its doors, the Z-machine was still widely available.

So when Graham Nelson started working on the first version of Inform in the early 1990s, he made a very smart decision: He wouldn't try to write his own game delivery system from scratch. Instead, he'd create an IF programming language that would produce game files that could be loaded into the Z-machine. Inform was written in such a way that its compiler would create Z-machine-compatible game files. These could then be played by anyone who had a Z-machine on a disk. This was one of several factors that insured the success of Inform.

Computers in those days had very little memory compared to computers today, so the Z-machine had to be small and efficient. It could load and run several different file types, but the most common had names that ended with .z5 and .z8. A .z5 file could be as large as 256Kb (yes, that's kilobytes, not megabytes), while the larger .z8 format could be used for games that needed to be as large as 512Kb.

Today, the Infocom-era Z-machine is ancient history. If you still have an Atari or Kaypro computer in working condition, and a Z-machine interpreter for that computer, you could play a game written today in Inform, as long as it was compiled as a .z5 or .z8 file. But who owns a Kaypro anymore? Today, several newer IF interpreters have been written that are compatible with the original Z-machine game file format. These have names like Frotz and Nitfol (both of which were magic words in one of the Zork sequels).

In order to allow Inform authors to write even larger games, Andrew Plotkin created the Glulx game format. Glulx games can be much larger, and can include sounds, graphics, and multiple sub-windows within the main game window. Glulx games can be played on any modern personal computer, since Glulx interpreters exist for Macintosh, Windows, and Linux. However, Glulx games are mostly too large to play on cell phones and other hand-held devices. Z-machine interpreters are available for some popular hand-helds.

Writing Your First Game

There are no rules at all that dictate what you can put in a game. One of the great things about interactive fiction is that you're free to write whatever sort of story you'd like, and put whatever story elements in it you think would be fun.

But if you hope to create a game that you can share with other people, you may want to think carefully about what people will be able to see and do when they play your game. Here are a few tips that may help you come up with a better game. If these tips don't make sense now, come back to the list in a few weeks — it may make more sense after you've done some writing.

1) Players won't be able to read your mind. If you want them to know about something (such as which directions they can go when they're in a certain room, or the fact that a time bomb is ticking), you need to write some output text that will tell them.
2) When creating objects, always add all of the vocabulary words you can think of that players might use to refer to the objects. (See Chapter 3 to learn how to do this.)
3) When creating new actions that will work on your objects, always add all of the verb synonyms you can think of. Forcing the player to guess what verb you had in mind is considered extremely poor form. (The way to do this is explained in Chapter 4.)
4) After writing a few paragraphs, click the Go button to compile your game and test your work. Test all of the silly commands you can think of, and watch how your game handles the commands. Then rewrite and test some more.
5) Write an intro to your game (using a When Play Begins rule — see Chapter 8) that gives players a clear idea about two or three things: Where the story takes place, what character they're supposed to be, and what that character is trying to do.

Inform contains a great mass of rules that will allow your game to do some sophisticated things automatically. The game can, for instance, construct sentences that you never wrote, in order to describe situations that may come up during the story. The rules that are included in I7 are called the *standard library* or *standard rules.* You can use them without knowing anything about them. In fact, you will be using them in every game, unless you explicitly switch some of them off. Just about every rule in the library can be individually disabled (switched off) if it produces results that you don't want. Switching off portions of the library is an advanced programming topic, however; we won't have much to say about it in this book.

From the Top

To start your game, you need to create at least one room, where the story will start. Chapter 2 of this book is all about how to create rooms. Before you start writing your first room, though, you may want to do a couple of preliminary things. I recommend starting your game as shown below. First, the game should have a title and a byline. (These will be created automatically by Inform when you start a new project, but you can change them at any time just by editing the first line in the Source page.) After the title and byline, skip a line and write a Use sentence, like this:

```
Use American dialect, full-length room descriptions, and the serial comma.
```

Inform was written by an Englishman, so it uses British spellings for a few words unless you instruct otherwise. “Use American dialect” switches the spelling from British to American.

When you use full-length room descriptions, the player will read the complete description of each room each time she enters the room. If you don’t use full-length room descriptions, the room description will be printed out in full only the first time the player enters a room; after that, the room name will be printed more or less by itself. (Any movable objects or people in the room will still be mentioned.) The player can switch full-length room descriptions on and off using the VERBOSE and BRIEF commands, which are built into Inform, so telling Inform that you want to use full-length room descriptions only controls how your game will start out, not what may happen after that. If the solution of a puzzle relies on the game being in verbose mode, but the player has switched to brief mode, you’ve written an unfair puzzle!

The serial comma is the final comma in lists of items. If the serial comma is switched on, your game might report this:

```
You can see an apple, a pear, and a banana here.
```

If the serial comma is not being used, the output is just slightly different:

```
You can see an apple, a pear and a banana here.
```

Most games also need an introduction — some text that will “set the stage” for the story. You can create an introduction by writing a When Play Begins rule, perhaps something like this:

```
When play begins, say "Lord Triffid has invited you to spend your week's vacation in his castle. Upon
arriving, though, you begin to feel a bit uneasy. Perhaps it's the bats flying in and out of the attic
window that put a damper on your mood, or perhaps it’s the sound of barking, snarling guard dogs...."
```

The Detail Trap

When writing your first interactive fiction, there’s a pitfall you may want to watch out for — the detail trap. Let’s suppose you want your story to start in the kitchen of the main character’s house. So you start putting things in the kitchen — appliances and so on. (By the way, **p. 8.5** of the *Recipe Book*, “Kitchen and Bathroom,” has some great tips on how to make appliances.) Now, a kitchen is a complicated place! The stove can be switched on, and touching it when it’s switched on will burn you. The hot and cold water taps in the sink can also be turned on, and when one of them is on, there’s some water that the player might want to interact with. In the refrigerator is some moldy cheese, so you need to figure out how Inform handles the smelling action.

And so on. Two months later, you’re still trying to work out the details of a realistic kitchen. (What if the glass in the cupboard is half-full of water instead of completely full? If you empty a half-full glass on the floor will it only make a small puddle instead of a large puddle?) Meanwhile, your story has gone nowhere, and your enthusiasm for interactive fiction has taken a nosedive. That's the detail trap.

A better approach, I've found, is to start by creating a bunch of rooms and putting perhaps a couple of major scenery items in each room. (Scenery is discussed in Chapter 2.) Then create a few simple puzzles. Then add one or two important characters — but only in a basic way. Don't worry about the nuances of conversation yet. (Characters and conversation are covered in Chapter 5.)

I like to write descriptions when I'm starting out, because I like to get a concrete feeling for the places and objects in the story. To start with, though, write basic descriptions of rooms and important objects without worrying about how the objects may change during the course of the game. You can always edit the descriptions later to allow them to change dynamically, or to take account of details that are added to the code later.

There's a lot to learn in Inform. So start with simple things and build up your game in an orderly way. Put off the complex and tricky stuff for a later stage in the development.

Title Trickery

We're going to take a little detour into more sophisticated Inform programming here — nothing that's tricky to write or understand, but we'll use a couple of Inform's deeper features without bothering to explain them. The reason we're going to do it here is because of what happens at the very beginning of the game.

After your intro, Inform will print the banner text. (Don't confuse the banner text, which is printed only once, with the status line, which looks like a banner and usually runs across the top of the interpreter window throughout the game.) If you've given your game a title and subtitle, they'll appear in the banner text, which might look something like this:

```
A Screw Loose
A Digital Dalliance by Jim Aikin
Release 1 / Serial number 090829 / Inform 7 build 5Z71 (I6/v6.31 lib 6/12N)
SD
```

Providing the release number, serial number, and so on is both sensible and a courtesy to players. Among other things, it will help you keep track of different versions of your game, if you release more than one version. But there may be games in which displaying this text is not desirable. You may want to replace the default banner text with your own text, or suppress it entirely. To do this, add code like this near the top of your game:

```
Rule for printing the banner text: say "I'm a lumberjack and I'm okay!"
```

Whatever you write will be printed out as the banner. Instead of using a "say" phrase as shown above, you can write "do nothing" here, which will suppress the banner text entirely. If you're replacing the banner text, you might want to include your own version number. (Inform's version numbering doesn't allow decimal points, which makes it a bit nonstandard in the computer world, so writing a banner text such as "A Screw Loose, version 1.01" might be worth considering.)

Writing a new rule for printing the banner text, however, will *also* change the banner text information when the player types the VERSION command. The release number, serial number, build number, and compiler number will be unavailable. This is less desirable. If you want to suppress the banner text at the beginning of the game but keep the information available in response to the VERSION command, you can do it this way:

```
The display banner rule is not listed in the startup rulebook.
```

This will eliminate the opening banner but leave the version information intact, so the player will be able to display it with the VERSION command. Substituting your own banner text when play begins while keeping the existing text in response to the VERSION command is tricky, so we won't get into it here.

Wondering how to put a cool subhead underneath the game's title? That text is called the story headline. You do it like this:

```
The story headline is "An extraordinary and disturbing coincidence".
```

Before we move on, we need to pause for a quick cautionary note: As of version 5Z71, the title of an Inform game can't contain an apostrophe at the end of a word. (This problem may be fixed in a future version.) The reason is a bit complicated. As explained on p. 215, Inform is somewhat smart about the English convention of using single-quotes inside of double-quotes. If it sees a word like this — goin' — in text that you've written for output, it will think the apostrophe is a closing quotation mark, and the output will be — goin" — . This is not desirable, so Inform also provides a way to get around it. You can produce an apostrophe by putting it in square brackets, like this — goin['] — . Unfortunately, this substitution trick won't work with the game's title, so titles like "Goin' Home", "George 'n' Andy", and "Moses' Inspiring Adventures in the Sinai" simply won't look right when printed out by the interpreter software that runs your game.

Telling a Story

Maybe the most basic way to look at a story — any story, be it interactive, written on paper, or told out loud — is that a story is about a person who has a *problem*. The reason for looking at stories this way is simple: If the main character in the story doesn't have a problem, the story will be extremely boring. When we read a story, we want to enjoy the suspense of wondering how the lead character will solve the problem, and then at the end we want the satisfaction of seeing how the problem was resolved. In interactive fiction, the player *is* the lead character, which can add to the suspense.

The problem in a story can be as small as finding a lost kitten, or as large as saving Earth from alien invaders. As long as the problem is emotionally important to the lead character, it will work in a story.

If the problem is too easily solved, the reader (or player) will feel cheated. So the author needs to make sure the problem is not only important to the character, but not too easy.

In interactive fiction, solving the main story problem usually means solving a variety of puzzles. This *Handbook* has a whole chapter (Chapter 6) on designing puzzles.

Managing Your Project(s)

It’s a good idea to always save your project to some specific folder on your hard drive. In Windows, for instance, this would probably be My Documents > Inform > Projects. If you care about your creative work (and you should!), it’s a very good idea to *back up* your project to a separate location after you’ve done any new work on it. USB memory “drives” are cheap and convenient. Always use a separate *physical* location for backup, not just a different partition on the same physical hard drive. The point of making a backup copy is to protect you against the possibility of a hard drive crash or system failure.

What’s a WIP?

As you read discussion of interactive fiction programming in Internet forums and newsgroups, you’ll often find somebody talking about “my WIP.” This is an abbreviation for “work in progress.” Large games can be “in progress” for months or even longer. As long as you’re still working on an unfinished game, or even thinking about it once in a while, it’s a WIP.

Every few days, I like to save a project using a new, numbered filename. After working on Flustered Duck 05 for a few days, I’ll use the Save As command and save the project as Flustered Duck 06, and so on. (The final release version of “A Flustered Duck” was project version 21). Here’s why that’s a good idea: If you should change your mind about a design decision that you’ve made, or if you should accidentally delete or change something without meaning to, you can open up an older version of your project and copy a portion of the source code from the old version into the new one. Doing this actually saved me from a serious problem when I was writing this *Handbook.* I recommend it highly.

All About Bugs

When a computer program doesn’t do what it’s supposed to do, it either does the wrong thing or, worse, stops working entirely. This type of problem is referred to as a *bug*. According to legend, one day the scientists who were trying to use one of the very earliest computers kept getting mysterious errors. After hours of frustration, somebody thought of opening the box of circuits and looking inside. A dead moth was lying on a circuit board, making an electrical connection where no connection was supposed to be. After that, the scientists started saying that any mysterious behavior in their programs was caused by “a bug.”

Finding and fixing bugs is a huge part of computer programming. Fortunately, most of the bugs you’ll run into don’t involve dead insects! (Although that might make an interesting puzzle....) Most bugs today are caused by errors in the instructions (the source code) that the computer is trying to run.

We all write source code that has bugs, so you may as well get used to it. Most bugs are easily found and easily fixed. A few of them may drive you crazy for hours at a time. It's all part of the process.

Inform bugs come in four different varieties.

When you click the Go! button to tell Inform to compile your source code into a game, the compiler may encounter errors. You may have written things that Inform doesn't understand. Instead of producing a game, Inform will print out an error message, or a bunch of them. Usually these messages will give you a pretty good idea what you need to fix — but sometimes the compiler can't quite guess what the real problem is, so you may need to try a bunch of changes until you find something that works.

In the second type of bug, when you click the Go! button, your game may compile successfully and appear in the right-hand window, ready for you to try out. But the game may not behave in the way you expect. An object that you meant to put in plain sight in a room might not be anywhere in the game, for example. Inform can't find these bugs for you. The only way to find them is by testing and retesting your game while writing it. Try out a bunch of different commands, including silly ones, just to see what happens. (This is called "trying to break it.")

The third type of bug is called a "run-time error." In a run-time error, your game tries to do something that it can't do. Instead of producing some type of normal output, the game will produce an error message. Run-time errors can happen, for instance, when your code tries to refer to an object that doesn't happen to exist. When you run into a run-time error, look closely at the error message in the game panel, and then at the code that is being used by the command that triggered the error.

The biggest run-time errors are those that cause the game to freeze or quit unexpectedly. Fortunately, these are fairly rare. But don't be surprised if it happens — just go back to the code that was being run in response to your last command. Nine times out of ten, that's where the problem will be found.

If you're having trouble finding a bug, a useful technique is to "comment out" a section of your source code that you think might be causing the problem. In Inform, comments are surrounded by square brackets [like this]. Any text that is within square brackets will be ignored by the compiler — unless it's within double-quotes. Within quotes, square brackets have a different purpose, which I'll explain a little further on.

By commenting out blocks of code, you can test a game both with and without selected features. This will help isolate a trouble spot.

The fourth type of bug can look like any of the first three. Because Inform itself is still being developed, the compiler undoubtedly has a few bugs of its own. If you encounter something weird when you seem to be doing everything right, it could be a compiler bug. 90% of the time it won't be — it will be your code. But compiler bugs do exist. If you should run into something that looks like a compiler bug, your first step should to post a message to the newsgroup rec.arts.int-fiction (see p. 36) asking for confirmation of what you've found. If it is indeed a compiler bug, you'll want to file a bug report using the form provided on the Inform website (go to http://inform7.com/contribute/report/).

Testing Your Game

Testing a game, or any other piece of software, happens in two stages. In the first (alpha) stage, you test your work as you're developing it. The best way to do this is to write the game one small piece at a time. For instance, when you add a couple of new rooms (as explained in Chapter 2), click the Go! button to run the game and try walking back and forth from the new rooms to the rooms that were already in place. Even with something as simple as adding rooms, it's possible to make a mistake in the compass directions, so you need to test your work.

If you add half a dozen rooms, a dozen pieces of scenery, and a machine the player can operate and *then* run the game, you're far more likely to miss a bug than if you test a little bit at a time. Worse, you may find yourself staring at a screen full of compiler error messages and have to spend an hour figuring out where your work went astray.

But even when you test as thoroughly as you possibly can while developing the game, you will miss dozens of awful bugs. This is a promise.

In the second (beta) stage of testing, you enlist the aid of a few industrious volunteers, who *beta-test* your game before it's released and send you reports of any bugs they spot. Good beta-testing won't transform an awful game into a great one, but it can definitely turn an awful game into an okay game, or turn a decent game with deep problems into a great game.

Always thank your beta-testers for sending you their bug reports!

Encourage your testers to use Inform's handy transcript feature (not to be confused with the Transcript in the Inform IDE). At the beginning of each play session, they should type the command TRANSCRIPT. This will open a file dialog box in which they'll be able to specify a name for a script (.scr or .log) file. This file will capture everything that happens in the play session. They can then attach the transcript file to an email and send it to you. In effect, you'll be able to "watch over their shoulder" while they play the game. This is incredibly useful — you'll learn a lot about what commands they try to use and what objects they're interested in examining.

Note that this feature doesn't work in the Inform IDE. It only works when the game is loaded into a separate interpreter.

Some IF interpreter software may wipe out a running transcript if your game ever clears the screen. You need to test this in several interpreters (and if possible on both Mac and Windows interpreters) before sending the game to your testers. Or ask them to test it, and tell them how to do so.

Traditionally, IF beta-testers include comments in their transcript by starting a command line with an asterisk. For instance, you might find a line in the transcript that reads like this:

```
>* The room description mentions the vase on the pedestal, but I already
broke the vase.
```

By default, Inform will respond to a line like this by saying, "That's not a verb I recognize." While not an actual problem, this message soon gets annoying, since the tester is, after all, doing something sensible. What we'd like would be for Inform to respond with something like, "Comment noted." Fortunately, this is easy to fix. Copy the following code into your game:

```
Commenting is an action out of world applying to one topic. Understand "* [text]" as commenting.

Carry out commenting:
	say "Comment noted."
```

The details of this code (the definition of the action and the use of a Carry Out rule) are covered in Chapter 4. A slightly more flexible way to get the same result is shown in Example 397, "Alpha," in the Documentation. However, the regular expression matching used in that example doesn't match the asterisk character. To allow the tester to use asterisks, as shown above, we would need to edit the code in the example slightly, like this:

```
After reading a command (this is the ignore beta-comments rule):
	if the player's command matches the regular expression "^\p" or the player's command
matches the regular expression "^\*":
		say "(Noted.)";
		reject the player's command.
```

With this code in place, your testers can flag their comments with a *, !, ?, or : (or, for that matter, a ??? or !!!), either in a way that signals what type of comment they're making, or just for fun.

Using the Debugging Commands

Inform includes a good set of commands that can be used for speeding up your testing process and tracking down bugs in your game. These commands are available only during the development process; when you release your finished game, other players won't be able to use them. These commands don't actually find bugs; what they do is help you see what's going on in your game while it's running, and/or take shortcuts that will speed up the testing process.

The commands you can use while your game is running in the Inform program include ACTIONS, RULES, PURLOIN, GONEAR, and SHOWME. Also, for testing a series of commands quickly, you can write a TEST ME script. The Replay button can be used to rerun your most recent sequence of commands, which is a very useful way to find out if you've succeeded in fixing a problem that you had in the last run-through. The Skein window contains all of your earlier run-throughs, and you can use it to replay just about any series of commands you used earlier.

With the PURLOIN command, you can instantly "grab" any object in your model world. Objects in distant rooms and locked containers can be purloined — and you can also purloin things that can't be picked up at all in the ordinary way, such as people and scenery. There's no reason to purloin people or scenery, other than for fun. But by purloining the rusty iron key that's hidden in the oak cask in the wine cellar, you can quickly test whether the key works to unlock the door of the princess's chamber. You don't need to go down to the wine cellar and break open the cask; just PURLOIN the key.

The GONEAR command provides instant transportation to any location in your game. With a large game, one that includes dozens of rooms, GONEAR will save you hours of work. Just pick an item of scenery in the room you want to be transported to, and GONEAR the scenery item.

The ACTIONS command causes Inform to print out the action that's being triggered when you type a command. This can be useful if you aren't getting the results you expect from a command.

The SHOWME command (see **p. 2.7** of the Documentation, "The SHOWME command") prints out the current state of any object in the model world. SHOWME SHOES, for instance, will print out all of the data associated with the shoes object. When you start writing objects that can get into several different states (for instance, a goblet of wine that can be poisonous or safe), you can use SHOWME GOBLET after dropping the tablets into the goblet to see if the goblet has switched to the state that it's supposed to.

If you're using scenes in your game (see Chapter 7), the SCENES command can be used to show which scenes are currently running.

The TEST ME command (see **p. 2.8** of the Documentation, "The TEST command") gives you a quick way to run through a series of commands. To start with, define the series of commands you want to use in your game, like this:

```
Test me with "open suitcase / unlock suitcase with rusty key / open suitcase / search suitcase / take
dynamite / light match / light dynamite with match / z / z / z".
```

After compiling your game, when you type TEST ME the game will automatically run through this entire sequence of commands, in order. When you're testing complex puzzles over and over, this will save you lots of typing. The word "me" here is used a lot, but it's not necessary. You could just as easily do it this way, writing two or more separate tests and then, if you like, writing a TEST ME command that nests some or all of the other tests:

```
Test suitcase with "open suitcase / unlock suitcase with rusty key / open suitcase / search suitcase".
Test dynamite with "take dynamite / light match / light dynamite with match / z / z / z".
Test me with "test suitcase / test dynamite".
```

Now the commands TEST SUITCASE and TEST DYNAMITE can be used independently, or they can be combined by typing TEST ME.

The RULES command can be used while testing your game to find out what rules Inform is using to process other commands. This is a fairly advanced technique — the first few times you try it you may not understand what you're seeing — but it can sometimes be very useful. If the game is producing an output that you don't want, you may be able to figure out what rule is producing that output by using RULES and then trying the command that isn't working correctly. But the RULES mechanism is not perfect: Sometimes the parser will reject an input before considering any of the rules in the main rulebooks. Here's an example:

```
>rules
Rules tracing now switched on. Type "rules off" to switch it off again, or
```

```
"rules all" to include even rules which do not apply.

>steve, take the crown
Steve has better things to do.
```

In this case, RULES doesn't tell you what rule is causing Steve to reject the player's command. To find the fix, I searched the Documentation (using the Search field) for the phrase "has better things to do". This took me immediately to **p. 12.4**, "Persuasion," where I found the answer. I'll jump ahead here and give you the answer, even though we won't discuss how it works until Chapter 5. What we need, if we want to change what happens in the game, is a Persuasion Rule:

```
Persuasion rule for asking Steve to try taking the crown:
	persuasion succeeds.
```

Another Way to Debug

One of the easiest traditional ways to debug a computer program is to add print statements. A print statement (which in Inform would be in the form of a "say" statement) prints a text output to the screen. You can put whatever you want in a print statement. For instance, you could do something like this:

```
To demolish the dungeon:
	say "Now entering the dungeon-demolishing code.";
	[other lines of code would go here...]
```

The "say" line doesn't do anything in your game; it just enables you to test that your code is actually calling the "demolish the dungeon" routine when you think it's supposed to.

Rather than just print out a bare "now this is happening" statement, you could write a testing "say" line that would give you specific information about other things that are going on in the game. Let's suppose, for instance, that you want the dungeon to be demolished only if the player is carrying the detonator — but for some reason it appears that the player is able to demolish the dungeon even while not carrying the detonator. In that case, you might want to check the player's inventory while printing out your debugging message:

```
To demolish the dungeon:
	say "Now entering the dungeon-demolishing code. The player is currently carrying [a list of things carried by the player].";
	[other lines of code would go here...]
```

The thing to be careful of with this technique is that you might accidentally leave a debugging print statement in the released version of your game! Erik Temple has suggested a neat way of dodging this danger, however. This uses a bit of inserted Inform 6 code, an advanced technique that is mentioned only briefly in this *Handbook,* on p. 257. But since we're on the topic of debugging, here's how to do it. First, add the following lines to your game, entering them carefully (the spaces are important):

```
To say debug:
	(- #ifdef DEBUG; -).
To say end debug:
        (- #endif; -).
```

Having done this, you can now rewrite any debugging print statements that you add to your game's code like this:

```
        say "[debug]Now entering the dungeon-demolishing code. The player is currently carrying [a
list of things carried by the player].[end debug]";
```

When you begin and end the quoted "say" line with "[debug]" and "[end debug]" as shown (being careful to put "[end debug]" *after* the period), the say statement will disappear when you use Inform's Release button to create the release version of your game.

Puzzles

Let's be frank: This *Handbook* is not going to tell you everything you might want to know about writing puzzles. For one thing, clever authors keep dreaming up new possibilities! But puzzles are such a big part of interactive fiction that a book on how to write IF using Inform can't neglect them entirely.

A few works of IF have been written that have no puzzles. In a work of this sort, the player wanders around, looking at things and/or conversing with characters, but there are no obstacles to movement or action. The whole of the story is available, and no special smarts or problem-solving are needed to read it. There's even an occasional competition, the IF Art Show, dedicated to this type of game.

Even in a story with no actual puzzles, the player may be able to make choices. These choices may affect the outcome of the story: The author might write five or six branching story lines, and the player might have to play the game a number of times to be sure of not missing anything important, or to understand what the author had in mind. But if the story is free of puzzles, we would expect that any of the choices would be easy to find. It would be easy to move down any of the branches of the story.

In most IF, though, the player has to exercise some brain power to do things that will move the story forward. The player who can't figure out what to do next is *stuck.* A player who is stuck can wander around in the world of the story for an hour, trying things that don't produce any results and getting more and more frustrated.

When the player does figure out how to solve a puzzle, there is usually a reward of some kind. A new object might become available (something the player will need to attack another puzzle), a new treasure be discovered, and a new room or region of the map open up. In games that keep score (see Chapter 7), the player should earn points for solving each puzzle. You may want to set up a scoring system that will award more points for solving the most difficult puzzles, and fewer points for solving the easier ones. Of course, players may not agree with you about which puzzles are easy and which are hard!

For details on the types of puzzles you're likely to find when playing IF, or likely to want to implement, see Chapter 6, "Puzzles."

Extensions for Inform

Many people in the Inform 7 community (which exists only on the Internet) have written *Extensions* for Inform. An Extension is a file containing code that can be used to do something specific — something that's not included in the standard version of Inform.

There are Extensions for creating tricky kinds of objects, such as liquids and secret doors; for producing smoother and more interesting conversations between the player character (PC) and non-player characters (NPCs); for constructing menus of hints that your players can consult; and for many other purposes.

Extensions are written using the Inform programming language — so technically, an Extension can't do anything you couldn't do yourself. But most of them were written by experts, they have been tested and debugged, and they come with documentation that tells how to use them. So why spend a week working out how to do something tricky when you can download an Extension and have it working in 15 minutes?

Extensions can be downloaded from the Extensions page of the Inform 7 website (http://inform7.com/write/extensions/). Before downloading your first Extension, you may want to create a folder called Downloaded Extensions in your Inform folder. Save all of your downloaded Extensions to this folder.

Important: After downloading the Extension, you need to install it before you can use it. Installing it makes the Inform program aware that it exists. To do this, go to the File menu and select "Install Extension...". In the dialog box that opens up, select the Extension that you've downloaded and click the Install button. Installing an extension copies it to a special Extensions folder, so there will now be two copies of it on your hard drive — one in your downloads folder and another in the Extensions folder. If you've already installed other Extensions by the same author, all of them will share a subfolder within the Extensions folder.

If you've done the installing correctly, the documentation for the Extension will now be included in the Documentation section of the Inform program. To check this, go to the table of contents for the Documentation, scroll down if you need to, and click on Installed Extensions. You'll find that they're listed alphabetically by the first name of the author.

Once the Extension is installed, using it in your game is easy. Near the top of your source code, just below the name of the game and the author byline, enter a line like this:

```
Include Plurality by Emily Short.
```

This format is required: You have to give the name of the author, and you can't put a comma after the name of the Extension.

If you're using Extensions, you may want to check the Inform website periodically to see if any of the ones you're using have been updated. Updates add new features or correct bugs.

Note: After opening an Extension in a new window in the Inform IDE, you can edit the code just the way you would any other Inform code. I *strongly recommend* that you not do this. Until you become an Inform power user, it's far too easy to create messy problems by editing an Extension. If you want to try editing an Extension — for instance, because you think it has a bug in it — make a backup copy of it first by copying the file and pasting it to a new folder on your hard drive. That way, if you don't get the results you're expecting, you can restore the original version without needing to download it again.

Where to Learn More

Once you've started writing with Inform, you'll find the built-in Documentation extremely useful. Pages in the Documentation that may not have made much sense to begin with will turn out, after a month or two, to be very clearly written and easy to understand. In particular, the more than 400 Examples are a huge resource — but to get the most out of them, you'll need to study the code line by line. The explanations that are included in the Examples are often brief and don't mention interesting features. After reading portions of the Documentation, you'll find the Search field in the IDE more and more useful. By typing a half-remembered phrase (such as "something new"), you can quickly locate a page that has the information you need.

If — no, make that "when" — you have questions about how to use Inform, you'll find the Internet newsgroup rec.arts.int-fiction a tremendous resource. (The newsgroup is often referred to by regulars as *r.a.i-f.*) If you post a question there, you'll probably be able to read an answer posted by an Inform expert within a few hours — or at worst, the next morning.

What's a Newsgroup?

Back in the early days of the Internet (before the World Wide Web, even), people posted messages to an Internet area called Usenet. Usenet is still around. It contains tens of thousands of newsgroups. A newsgroup is sort of like a Web-based forum: Anyone can post messages, which can be viewed by anyone else.

Your email program may have a page with which you can access newsgroups. However, you may have to pay extra to get access to a newsgroup server. Some ISPs (Internet Service Providers) include newsgroup access in a basic account, but many others don't. If you'd like help getting started with newsgroups, you can consult the article on this topic in Wikipedia.

You can also read newsgroups through Google Groups, using your Web browser. Access to Google Groups is free, but the user interface is frankly not good, and Google doesn't filter out spam, which means the message list tends to fill up with junk mail.

To get the most out of the newsgroup, a few simple procedures will help:

1) Put "I7" at the beginning of the subject line of your post. Inform 7 is not the only programming

system discussed on the newsgroup, and you want to guide the right people to your post by making it clear what system you're using.

2) Describe your problem as clearly as you can. Questions that are rambling or confusing will be less likely to lead to helpful responses than questions that are clear and concise.
3) Before posting, try to create a very short example game (no longer than 20 lines) that shows the problem you're having. This will make it easier for the experts to see what you're doing wrong — and in the process of creating the example game, you may figure out how to solve the problem yourself! (It often works that way, in my experience.)

The IF Forum (http://www.intfiction.org/forum/) gets less traffic than the newsgroup, but the Inform experts do seem to check it pretty regularly, so it's another good place to post questions. For more general advice on writing IF, the ifwiki (http://www.ifwiki.org) is a good place to start your search. A lot of information is available on the Internet, some of it tucked away in unlikely places, but the wiki has links to most of it. Emily Short's interactive fiction pages (http://emshort.wordpress.com) also have lots of useful resources.

Chapter 2: Rooms & Scenery

As explained in Chapter 1 (see "What Happens in a Game"), the model world in which your story takes place will be divided neatly into rooms. For the purposes of IF, a "room" might be not a kitchen or living room; it might be a football field or a phone booth.* The same word is used in each case. Some people feel that the word "room" is misleading, but we have to call the places where things happen by some name or other, and the word "room" has been used for this purpose for a long time.

While characters (including the player) can travel from room to room, for most purposes we can think of each room as a sealed box. When the player is in a given room, she will be able to see, touch, and use the things that are in that room. But everything in every other room will be invisible and out of reach. In fact, if the player tries to refer to something that's not in the current room, the parser (the software in the game engine that interprets what the player types) will usually pretend that the object the player referred to doesn't exist.

This is usually what's needed in a game, but sometimes having a room operate like a sealed container won't give you the results you're hoping for. If so, you'll find that there are two or three ways to get around this limitation. If the player is at the north end of an open field, for instance, you might want him to be able to see (but not touch) the things that are at the south end of the field. Later in this chapter, in the section "The Great Outdoors," we'll suggest some ways to do this. There are also some exceptions that come into play when you're designing a conversation system to let the player talk to other characters: You can easily allow ASK BOB ABOUT THE JEWELS to work as expected even if the jewels themselves are not in the room. In most situations, though, an object has to be in the room with the player in order for the player to do anything with it or even see it.

This concept is one of the most basic in interactive fiction. It's called *scope.* An object is "in scope" if the player character can see it; otherwise it's "not in scope." The parser is responsible for enforcing the rules of scope. If the parser pretends not to know about a given object when the player refers to it, it's because the object is not in scope. This will most likely happen because the object is in a different room. But it can also happen because the object is in a closed container in the room, or because there's not enough light in the room to see anything.

* A phone booth could also be created as an enterable container within the main room — see the section on "Enterable Containers & Supporters" in Chapter 3 of the *Handbook*.

By convention, everything in a room is assumed to be equally within reach of the player character, with a couple of exceptions. As we'll see in Chapter 3, objects can be placed in locations that are out of reach, so that the player can look at them but not touch them. In addition, the player character may be seated on a chair or lying in a hammock in the room, in which case she will have to STAND UP before interacting with objects. In general, though, rooms have three standard characteristics: Everything in the room is equally available, the player character is not facing any particular direction, and things that are dropped end up on the floor. All of these conventions can be changed, but experienced IF players will take them for granted, and there's usually no need to concern yourself with them — especially not when writing your first game.

Creating Your First Room

Every Inform game has to have at least one room. Creating a room is easy; you do it like this:

```
The Forest Path is a room.
```

This simple sentence produces, in fact, an entire Inform game (though not a very interesting one). If you create a new game, type this sentence into the Source, and compile the game by clicking the Go button, you'll find yourself in a (featureless) room called Forest Path.

It's usually a good idea to capitalize the words in each room name. This is not required, it's just a good habit to get into because it will produce a more professional-looking game. You can add "The" before the room name if you like. Inform will normally strip out the "The" before printing the room name in the game's output, so a room you call The Forest Path will appear in the game as Forest Path. You can override this if you want to, though, using the **printed name** property, like this:

```
The printed name of the Forest Path is "The Forest Path".
```

Notice the punctuation here: The period is *outside* the quotation mark, not inside. This is important. When Inform sees a period just before a close-quote, it will add an extra blank line in the game's output each time it prints the text. We don't want that to happen with a room name (and we especially don't want it to happen with some room names but not others, as that would make a mess of the output). So putting the period outside the close-quote is the right thing to do.

You'll almost always want to give each room its own **description**, to give the player some idea where she is. The description is text to be printed out, so you write it in double-quotes, like this:

```
Forest Path is a room. "Tall old trees surround you."
```

A room description can be as long or short as you like. It always has to end with a period (or with a question mark or exclamation point, though those aren't used much in room descriptions.) In this case we want the period *inside* the quotation mark, because we want Inform to put a blank line after the description and before whatever is printed next.

How Not to Describe a Room

When writing room descriptions, there are two traps to watch out for. First, try to avoid starting every room description with "You are in..." or "You are standing in...." This gets boring very quickly. Just describe the setting. Second, avoid describing the room as if the player has just arrived from any particular direction, or is facing any particular direction. As you're first writing the game, you may naturally tend to assume that the player has arrived in the dining room from the front hall — but later in the game, the player may arrive in the dining room by returning from the kitchen. A good room description will give the same impression no matter where the player arrived from.

In describing the room, you'll almost always find yourself mentioning things that can be seen in the room. They may be important to the story, or they may be there simply to add color and atmosphere. The things that are mentioned in the room description are *scenery*.

In general, it's a very good idea to add a separate scenery object (see the next section) for each thing that's mentioned in a room description. The people who play your game don't start out knowing what's important in the story and what's irrelevant, so players usually run around examining any object that's mentioned in the game's output — or trying to examine it. If the player tries to examine something that isn't in scope, the parser will respond, "You can't see any such thing." If the thing has just been mentioned in the room description, this message is annoying, and your players will soon decide you're not trying very hard. Imagine this situation in a game:

```
Forest Path
Tall old trees surround you.

>x trees
You can't see any such thing.
```

In the early days of interactive fiction, this type of response was considered normal, but at that time computers had much less memory. Game authors had to make the most of a tiny amount of text and a very small number of objects, so none could be wasted on irrelevant scenery. Today, failing to put the scenery into your rooms is considered poor form. In the next section, we'll look at how to add scenery.

It's important to write room descriptions that give players a clear idea which directions are available for travel. That is, if the player can travel east, northeast, or west to reach other rooms, the room description should mention these directions and perhaps give some vague idea what lies in that direction:

```
Forest Path
Tall old trees surround you. The path runs roughly north and south from
here, and a little side path runs off through the bushes to the northeast.
```

Failing to mention exits in the room description was used as a puzzle in some early games, but today this type of thing is generally considered rude and crude. Fortunately, there are Extensions (such as Exit Lister by Eric Eve) that will put the room's exits into the status line at the top of the game window. Using Exit Lister or something similar is a nice courtesy — but it's still a good idea to write a room description that describes all of the exits in a clear way (unless one of them is hidden as a puzzle). Exit Lister also provides a utility command for the player, EXITS, which will list the available exits to refresh the player's memory.

Inform's routines for printing out room descriptions are actually quite complex. After printing out the description you write, Inform will automatically mention anything in the room that it considers interesting — mainly people and visible objects. It uses certain rules for deciding how (or whether) to mention these things. If you want to customize the way room descriptions are printed out, see "Room Descriptions" at the end of this chapter.

Scenery

Inform allows us to create many **kinds** of objects. The standard library includes containers, mechanical devices, doors, people, and so on. (See **Chapter 4** of the Documentation, "Kinds," to learn more about kinds.) The word "kind" is a technical term in Inform: It's used to define new types of objects when the game will include several of them, and we want them to behave in similar ways. For instance, if a puzzle involves spelling words with alphabet blocks, we might do this:

```
An alphabet block is a kind of thing.
```

After writing this, we could write rules that would apply to all alphabet blocks, and Inform would know what we were talking about.

The first type of object we're going to meet in this book is scenery. To make matters a little more confusing, however, the word "scenery" in Inform does not refer to a *kind* of object; it's a property that can be applied to almost any object. A thing (the basic kind) can be scenery, a container or supporter can be scenery, a door can be scenery, a device can be scenery, and so on.

If we don't say anything more specific when creating scenery, Inform assumes that the object we're creating is just an ordinary thing, not a device, door, or container. It also assumes that scenery is fixed in place: that it's not something the player can pick up and carry around.

When your game is constructing a room description to print out, it will mention any ordinary objects that are lying around in the room, but it won't mention scenery objects (unless the object is a supporter that has something else sitting on it; see p. 86 for more on supporters). Inform assumes that you mentioned the scenery objects in your own room description when you wrote it, so be sure to do so.

We can add scenery to our forest path like this:

```
The Forest Path is a room. "Tall old trees surround you."
The tall old trees are scenery in the Forest Path. The description is “Ancient oaks stretch out their
heavy branches overhead, blocking the sun.”
```

Now the player who types X TREES will be able to read a description that adds detail to the scene. Notice that here I’ve started the sentence by saying

```
The description is [… and so on …]
```

The phrase “The description is” is optional with scenery and rooms, but it’s *required* with objects that can be picked up and moved around (as described in Chapter 3). If you don’t use “The description is” with scenery, you should be sure to always put the description in the very next sentence after you’ve created the scenery object. This code works exactly like what was shown above — but only with scenery, not with ordinary things.

```
The tall old trees are scenery in the Forest Path. “Ancient oaks stretch out their heavy branches
overhead, blocking the sun.”
```

If you try out this code, you’ll soon discover two problems. First, you can X TREES, X TALL TREES, or X OLD TREES, but you can’t X OAKS or X ANCIENT OAKS. The parser will reply, “You can’t see any such thing,” which is fairly silly. It happens because Inform never looks inside of double-quoted text to see what words you used there. Second, if you should try something like TAKE TREES, the parser will complain, “That’s hardly portable.” The word “trees” is plural, so the correct output would be “Those are hardly portable.” But the parser doesn’t know that the trees object is plural. We have to help it out a little.

To solve the first problem, we’ll add an Understand rule to the trees. You’ll soon find that most of the objects you create in your games will need Understand rules. Their main purpose is to add vocabulary — extra words that the player can use to refer to the objects.

To solve the second problem, we need to make sure the parser understands that the tall old trees object is **plural-named.** There are two ways to do this. We can do it explicitly, by adding the sentence, “The tall old trees are plural-named” to our source. Or we can do it implicitly, by changing the sentence where we create the trees so that it refers to “Some tall old trees”. The second method is easier. Inform knows that when things are created using the word “some”, they’re plural-named. When we put it all together, the trees will look like this in our source code:

```
Some tall old trees are scenery in the Forest Path. “Ancient oaks stretch out their heavy branches
overhead, blocking the sun.” Understand “ancient”, “oaks”, “heavy”, “branches”, “branch”, and “tree” as
the tall old trees.
```

Notice that when we add vocabulary words to the trees object, we have to put the commas that separate items in the list of words *outside* the quotation marks.

This usage of “some” to create a plural-named object is convenient, but there’s a potential problem to be aware of. In English, some nouns are “collective.” Examples would include things like sand and water. If we write “Some sand is scenery in the Beach,” Inform will be confused into thinking the sand is plural-named. If it needs to construct a sentence that includes the sand object, it will say “The sand are....” To avoid this, we need to create the sand using “the”, not “some” — and then we need to tell Inform to refer to the sand as “some sand” when it needs to construct a sentence, not as “a sand”. Here’s how to do it:

```
The sand is scenery in the Beach. The indefinite article of the sand is "some".
```

The term “indefinite article” refers, in English, to the words “a” and “an”.

Once we’ve created the tall old trees as a scenery object, we can refer to the object in our source code either as “tall old trees” or just as “trees”. The compiler will understand either form. However, it’s a good idea to get in the habit of always using the full names of objects when writing your game. If you use the short forms of names, Inform will usually understand what you meant, but it’s possible to end up with code that includes hard-to-find bugs.

The reason is this: You may have several objects in your game whose name ends with the noun “trees” — the tall old trees in the forest, the pear trees in the orchard, and the shoe trees in the closet. If you just say “trees” in your source code, the compiler will try to figure out which object you meant, and it will usually get it right. But once in a while it will get confused. The result could be disastrous: The player might try to pick up the shoe trees and end up carrying around the whole forest or the orchard by mistake.

For the same reason, it’s a good idea to get in the habit of naming every object with at least one adjective in addition to the noun. Odd things can happen if you have one object called the beach ball and another object that’s simply called the ball.

In case you’re wondering — no, it’s not possible to refer to the tall old trees in your source code as “the ancient oaks”. Words in Understand rules are strictly for the player’s convenience, not for the author’s use.

At the beginning of the game, players don’t know what’s important. So they’ll try out anything they can think of. With the tall old trees, we can expect the player to try CLIMB TREES (or more likely, CLIMB TREE, as you can’t very well climb several trees at once). By default, this will be the output:

```
>climb tree
I don’t think much is to be achieved by that.
```

There’s nothing wrong with this output, except that it’s boring. We can make it more interesting using an Instead rule:

```
Instead of climbing the tall old trees:
	say “Vicious flocks of sparrows and wrens dart down at you and peck at you with their sharp little beaks, driving you back.”
```

The actual result in the game is the same as before: nothing has happened. The player is still standing in the room called Forest Path. (And if the player tries X SPARROWS, there won’t be any sparrows in the room. We haven’t yet created any!) But the game is a little more interesting and fun than before. As you write messages like this, you might even start to wonder, what are those sparrows and wrens trying so hard to protect? This might suggest a puzzle that you can add to the game, such as scattering birdseed to distract the sparrows and wrens. After scattering the birdseed, the player might be allowed to climb a tree after all. But that type of complication will have to wait for a later chapter.

A Really Important Note!

As you read *The Inform 7 Handbook,* you'll find dozens of examples of game code that you can copy and paste directly into the Source page of the Inform program in order to try out the features that are illustrated in the code. These examples are all in blue type. Unfortunately, if the examples contain indented lines, just copying and pasting them from the *Handbook* into your game ***won't work***. You can select the text of the examples in either Preview (on the Mac) or Adobe Reader (in Windows) and copy it, but when you paste the text into the Inform Source panel, all of the indentation will disappear. (So will the blank lines between blocks of code.) And without the indentation, Inform won't understand the code.

To get the examples to work in Inform, you'll have to eyeball them in your PDF reader program, figure out where all of the indentations are (and exactly how many Tab keys each line is indented), and copy the indentations yourself, one line at a time. (See Chapter 8 of the *Handbook* for more on indentation.)

You can restore the blank lines by adding extra Returns, or not — it's up to you. Their presence or absence shouldn't change how the code works, but adding blank lines will make the code easier to read.

Also, if you block-copy from the end of one page of the *Handbook* onto the beginning of the next page, the page number will be copied along with the code. You'll have to delete it in Inform.

How Much Scenery Is Enough?

As noted in Chapter 1, in the section "The Detail Trap," it's easy to get sidetracked by trying to cram too much scenery into a room. If your story is set in a modern house, the house will probably have a bathroom. Should you add a toilet, a sink with faucets, a mirror, a bathtub with a showerhead, towels, and wash cloths to the bathroom? Probably not. It's a lot of extra work — and if you include these objects, the player will expect that manipulating them will have something to do with the story.

All the same, it's jarring, if the game includes a bathroom, to see this kind of output:

```
Bathroom
A typical bathroom. The door is to the west.

>x tub
You can't see any such thing.
```

```
>x sink
You can't see any such thing.

>x toilet
You can't see any such thing.
```

These responses destroy the illusion that the player is in a real room. A better approach is to create a single scenery object that stands in for all of the things in the room that aren't important:

```
Some fixtures are scenery in the Bathroom. Understand "sink", "toilet", "faucet", "mirror", "cabinet", "tub", "basin", "shower", "towel", "bath", and "mat" as the fixtures. The description is "The bathroom fixtures are not very interesting."

Instead of doing anything other than examining with the fixtures:
        say "You have more important things to do right now than fiddle with the bathroom fixtures."
```

This preserves at least a thin illusion that the bathroom is a real place, while directing the player's attention elsewhere.

Generally speaking, if an object is mentioned in the room description, it should probably be implemented as a separate scenery object. A "stand-in scenery object" like the one shown above would be a better choice for things that the player might naturally expect to be in this sort of room, but that aren't important to the game.

Distant Scenery

It often happens, as you write a room description of an outdoor "room," that you'll want to mention things that are far away — visible, but not something that can be interacted with. Inform has no built-in way to handle this situation, but we can easily create one. We'll start by showing the problem, and then show how to fix it.

```
The Forest Path is a room. "Tall old trees surround you. The path continues north and south. In the distance to the west, off among the trees, you can see a crumbling stone wall."

The crumbling stone wall is scenery in the Forest Path. "The wall is ancient and moss-covered." Understand "ancient" and "moss-covered" as the stone wall.
```

This code is fine, up to a point. The player can now examine the wall. But Inform has no idea that the wall is far away. The result, when the player tries doing things with the wall, is less than great:

```
>x wall
The wall is ancient and moss-covered.

>touch wall
You feel nothing unexpected.
```

```
>take wall
That's hardly portable.

>push wall
It is fixed in place.
```

Obviously, the player character wouldn’t be able to touch or push the wall, because it has been described (in the room description) as far away. To solve this problem, we’re going to create a new **property**. Objects in Inform have many properties. (You can view the properties of any object in the Game panel by typing SHOWME and the name of the object.) The printed name and description are properties, for instance. And creating new properties is easy. Let’s create an either/or property, distant/near. Every object in the model world will be either distant or near, but all of them will be near unless we say otherwise:

```
A thing can be distant or near. A thing is usually near.

Instead of doing anything other than examining to a distant thing:
	say "[The noun] [if the noun is plural-named]are[otherwise]is[end if] too far away."

The crumbling stone wall is scenery in the Forest Path. "The wall is ancient and moss-covered."
Understand "ancient" and "moss-covered" as the stone wall. The stone wall is distant.
```

After creating distant as a property, we can make the stone wall (or anything else) distant, just by saying that it is. Now the output will be a lot more sensible:

```
>touch wall
The crumbling stone wall is too far away.
```

The code above assumes that the distant thing will always be the first object the player refers to in an input. That is, it will be the *direct object* of the verb. In Inform source code, the direct object is referred to as the **noun**. If there’s another noun later in the command (usually it would be in a prepositional phrase), you would refer to it in your source as the **second noun**.

To be safe, we might also want to trap commands like PUT VASE ON STONE WALL. In this command, the stone wall is the second noun. To handle this, you can write:

```
Instead of doing anything when the second noun is a distant thing:
	say "[The second noun] [if the second noun is plural-named]are[otherwise]is[end if] too far away."
```

This code won’t work well if the command involves, for instance, talking to an NPC about a distant thing. ASK GUARD ABOUT STONE WALL would produce the output, “The stone wall is too far away.” Fixing this would take us into the realm of advanced Inform programming. If you’re writing a game where this becomes a problem, posting a message on rec.arts.int-fiction should get you steered in the right direction. You might also want to look at the Extension called Far Away by Jon Ingold.

Notice the use of the if-test, which is embedded in the output text of the instead rule. If we just wrote, “[The noun] is too far away”, distant things that were plural-named, such as geese winging across the sky, wouldn’t work quite right. We’d see this:

```
>touch geese
The geese is too far away.
```

With the geese, we might prefer to create a distant thing called the flock of geese, and not make it plural-named (since “flock” is singular). A herd of cows could be handled the same way. But you should get in the habit of using this type of if-test whenever you’re writing default messages that may have to apply to a number of different objects. If you install Plurality, an Extension by Emily Short, you’ll have a smoother way of taking care of this particular problem; the basic way is shown here mainly because this is a textbook. Learning how to do it yourself is part of the point of the book. If you include Plurality in your source code (see “Extensions for Inform” in Chapter 1, p. 35), this type of output is cleaner and easier to write:

```
Instead of doing anything other than examining to a distant thing:
	say "[The noun] [is-are] too far away."
```

For a more complex way of creating an outdoor environment that includes items in other locations, see “Indoors & Outdoors” in Appendix B.

Adding More Rooms to the Map

It’s possible to write a complete short game that takes place in a single room. But most games will need a number of rooms. The player travels from one room to another using standard compass directions — NORTH (or simply N), SOUTHWEST (SW), and so on.

Other methods of travel have been tried by various authors, and you may want to experiment with them at some point. In “Blue Lacuna,” by Aaron Reed, you can travel from place to place by typing the name of something that lies in the direction that you want to go. In a game designed like this, CASTLE will take you closer to the castle (or cause you to enter it, if you’re already close), and so on. Aaron has released this system as an Extension, so you can use it in your game without having to understand every detail … though it’s more complex than the average Extension, so you should expect to do some study and some testing in order to get it to work the way you’d like it to in your game.

If your story is set aboard a ship, you may want to replace N, S, E, and W with FORWARD, AFT, PORT, and STARBOARD. (**Example 40** in the Documentation, “Fore,” shows how to do this. **Page 3.26**, “Directions,” offers some other suggestions.) But for your first game, I’d suggest sticking with compass directions. All players know how to use them.

Inform makes it very easy to set up a map containing rooms that are connected by compass directions.

Pages 3.2 and 3.3 in the Documentation, “Rooms and the map” and “One-way connections,” explain how to do it, but we’ll take a quick look here, and suggest ways to work through problems that may come up.

Where Did It Go?

The numbering of the Examples in Inform’s Documentation may change in future versions. Likewise the numbering of the pages. However, the names of both Examples and pages should remain the same. If you can’t find a page cross-referenced in this *Handbook,* consider the possibility that the numbering has changed.

Once you’ve created your first room, you can create more rooms simply by describing the map to Inform, like this:

```
Forest Path is a room. “Tall old trees surround you. The path continues north and south from here.”

Canyon View is north of Forest Path. “The path from the south ends here at the top of a cliff, from which you have a spectacular view of a canyon.”

Haunted Grove is south of Forest Path. “The trees press close around you, moaning faintly and waving their branches in an unsettling way. A path leads north.”
```

This text will create a map with three rooms — from north to south, Canyon View, Forest Path, and Haunted Grove. Inform is smart enough to understand that if you say “Canyon View is north of Forest Path,” Canyon View must also be a room, because rooms and doors are the only things that can be related to other rooms using a compass direction. We haven’t told Inform that Canyon View is a door, so it must be a room.

Inform understands that the connections you have described run both ways. That is, because you’ve said Haunted Grove is south of Forest Path, Forest Path will automatically be mapped north of Haunted Grove. The player who goes south from Forest Path will be in Haunted Grove, and by going north from Haunted Grove the player will return to Forest Path.

The directions you can travel to leave a room are often called the room’s exits. But “exits” is not a term that Inform understands, unless you write some code or include an Extension that defines the word.

It’s easy to create room connections that are one-way, or that bend. We’ll show how to do that later in this chapter. A few map connections of this sort may be useful to make your world seem more real, but adding too many of them will just annoy players.

The room descriptions above all tell the player which directions they can travel in when leaving the room. This is a nice way to help the player, though it can make the writing seem a little clumsy. As mentioned earlier, you can download and install an Inform Extension (Exit Lister by Eric Eve) that will place the available compass directions in the status line at the top of the game window.

Capital Offense

In most situations, Inform is not fussy about whether you capitalize words in your source. But you can’t use a capital when mentioning map directions in your source (unless the direction is the first word in a sentence). Below are three code examples. The first two work, and produce exactly the same map; the third is a bug, and won’t compile.

```
The Lounge is a room.
The Dining Room is north of the Lounge.
```

```
The Lounge is a room.
North of the Lounge is the Dining Room.
```

```
The Lounge is a room.
The Dining Room is North of the Lounge. [Error!]
```

Page 3.2 of the Documentation, “Rooms and the map,” explains how to deal with a small but sometimes annoying problem in naming rooms and defining map connections: What if you have a room called Hut, and now you want to name a room South Of The Hut? This won’t work:

```
South Of The Hut is south of the Hut. [Error!]
```

Nor will this:

```
South Of The Hut is a room. South Of The Hut is south of the Hut. [Error!]
```

Either of these would work if Inform noticed the way we’re using capital letters, but it doesn’t. The solution is to use the word “called”:

```
South of the Hut is a room called South Of The Hut.
```

This sentence may look as if it was written by Gertrude Stein. (An avant-garde author of the early 20th century, Stein is most famous for the line, “Rose is a rose is a rose.”) This sentence could be read as “‘South of the Hut’ is the name of a new room. This new room is called (even though we’ve already said so, and now we’re repeating ourselves) South Of The Hut.” If Inform were to read the sentence that way, you’d have a new room called South Of The Hut, but it wouldn’t be connected to any other room in the map.

As you can see, Inform’s “natural language” syntax can occasionally be misleading or difficult for humans to read. The sentence actually makes sense if you read it a different way: South (a direction) of the Hut (a room we’ve already told Inform about) is a room (another room — a new one, this time) called South Of The Hut.

The same problem can arise if we want to call a room something like Inside the Stone House. Remember, Inform understands “inside” and “outside” (or “in” and “out”) as directions for travel. So we can’t do this:

```
Inside the Stone House is inside from Haunted Grove. [Error!]
```

Here’s how to do it:

```
Haunted Grove is south of Forest Path. "The trees press close around you, moaning faintly and
waving their branches in an unsettling way. A path leads north, and a house built of stone stands
here."

Inside from Haunted Grove is a room called Inside the Stone House. The description of Inside the
Stone House is "A small, dimly lit room. A doorway to the north leads deeper into the house."
```

Notice the phrase “The description of Inside the Stone house is”. This is not usually needed when we define a room, but it is needed here. If we simply start a new sentence with “A small, dimly lit room,” Inform will think we’re giving a description of Haunted Grove. But we’ve already given Haunted Grove a description, so we won’t be allowed to give it another one. The rule is, Inform looks at the *first* room in the previous sentence (Haunted Grove) and thinks it’s what we’re talking about if we just start a new sentence with a quotation mark. Since we want to write a description of Inside the Stone House, we have to help Inform understand what we have in mind.

The “The”

If you know a little about Inform programming, you might think that because the view-of-wall object created in the example below is privately-named and has a printed name, the code could just as easily read “View-of-wall is privately-named scenery...”, rather than, “The view-of-wall is privately-named scenery....” Since the name “view-of-wall” will never be used, why should the “The” make the slightest difference?

It does, though. If you omit the “The” when writing a paragraph that creates an object, Inform will assume that the object is proper-named. (An example of how this mechanism is designed to work would be if you create an object by saying, “Bob is a man in the Gazebo.”) The compiler will make this assumption even if the object is privately-named. As a result, substitution strings such as “[The noun]” won’t work properly. Normally, this substitution will produce “The wall” in the output, or “Bob”, as needed. The “The” (either capitalized or not) is suppressed with proper-named objects. The result: If you forget the “The”, you’ll see ugly outputs like “wall is too far away.”

The Great Outdoors

Earlier in this chapter, in the section “Distant Scenery,” we looked at how to make distant scenery items that could be examined but not interacted with in any other way. But it’s also possible that when you’re creating an outdoors area, you’ll want the player to be able to use commands like ‘look north’ to be able to inspect what’s in the distance. And if there are any objects in the game that are large enough to be seen from a distance, it would be natural to want the output of a ‘look north’ command to mention them.

In addition, once a large object has been brought to the player’s attention in this way, you might want the player to be able to examine it, even if the result is something like, “Professor Plum is too far away for you to make out any detail.” This requirement is a bit tricky to code, because as far as Inform is concerned, things that are in other rooms are not in scope, which means the player won’t be able to refer to them at all. Or rather, the player is free to refer to them, but the parser will pretend they don’t exist.

If you want to write a realistic outdoor setting, spend some time studying Chapter 3.4, “Continuous Spaces and the Outdoors,” in the Recipe Book. The examples there illustrate some powerful techniques. Here we’ll take a quick look at a couple of them. First, a slight revision of the “Distant Scenery” example. We’re going to create a couple of distant scenery objects. We’ll give them names that start with “view-of-” so that Inform won’t confuse them with the real objects in other locations.

```
Include Plurality by Emily Short.

A thing can be distant or near. A thing is usually near.

Instead of doing anything other than examining to a distant thing:
	say "[The noun] [is-are] too far away."

The Forest Path is a room. "Tall old trees surround you. The path continues north and south. In the distance to the west, off among the trees, you can see a low stone wall."

The winding path is scenery in Forest Path. "The path meanders from north to south."

The view-of-wall is privately-named scenery in the Forest Path. "The low stone wall appears to be ancient and moss-covered." Understand "wall", "low", "stone", "ancient" and "moss-covered" as the view-of-wall. The view-of-wall is distant. The printed name of the view-of-wall is "stone wall".

By the Wall is west of the Forest Path. "A crumbling stone wall runs from north to south here, blocking the way to the west. To the east, among the trees, you can see a path."

The crumbling stone wall is scenery in By the Wall. "The wall is ancient and moss-covered." Understand "ancient" and "moss-covered" as the stone wall.

The view-of-path is privately-named scenery in By the Wall. "The path is only faintly visible from here, meandering among the trees to the east." The view-of-path is distant. Understand "meandering", "path", and "trees" as the view-of-path. The printed name of the view-of-path is "path among the trees".
```

Next, we’ll allow the player to look in a direction:

```
Direction-looking is an action applying to one visible thing and requiring light. Understand "look [direction]" as direction-looking.

Carry out direction-looking:
    say "You see nothing unusual in that direction."
```

Next, we’ll create a response when the player looks in a direction where we’ve placed some interesting scenery:

```
Instead of direction-looking west in the Forest Path:
    say "There's a low stone wall off in the distance to the west."
Instead of direction-looking east in By the Wall:
    say "To the east, a path meanders from north to south among the trees."
```

But what if the player has dropped something large in the other location? Ideally, we’d like it to be mentioned when the player looks in that direction. This requires a few more steps. Near the top of the code for the game, we do this:

```
A room has some text called the containing-name. The containing-name of a room is usually "".
```

To test the functionality we’re going to add, we’ll need an object large enough to be visible from a distance:

```
A thing can be huge. A thing is usually not huge.
The beach ball is in Forest Path. The beach ball is huge. The description is "Red and white."
Understand "red" and "white" as the beach ball.
```

Next, we’ll give our outdoor rooms containing-names, and add a function that will use this property:

```
The containing-name of the Forest Path is "Lying on the path".
The containing-name of By the Wall is "Not far from the wall".

To scrutinize (R - a room):
    let L be a list of things;
    repeat with item running through things in R:
        if item is huge:
            add item to L;
    if the number of entries in L > 0:
        say " [The containing-name of R] you can see ";
        say "[L with indefinite articles].";
    otherwise:
        say "[line break]".
```

The final step is to revise the code for direction-looking so as to take advantage of the scrutinizing function:

```
Instead of direction-looking west in the Forest Path:
    say "There's a low stone wall off in the distance to the west.[run paragraph on]";
    scrutinize By the Wall.
Instead of direction-looking east in By the Wall:
    say "To the east, a path meanders from north to south among the trees.[run paragraph on]";
    scrutinize the Forest Path.
```

With these additions, when we go west to By the Wall and LOOK EAST, we’ll get this output:

```
>look east
To the east, a path meanders from north to south among the trees. Lying on
the path you can see a beach ball.
```

But one problem remains. The beach ball is being mentioned (because it’s huge), but the command X BALL will produce the output “You can’t see any such thing.” True, there’s not much to see, because the beach ball is far away, but it would be nice if the parser didn’t produce such a confusing output. To fix it, we would need to play with the scoping rules. Fortunately, Example 347 in the Documentation, “Stately Gardens,” which is part of the “Continuous Spaces and the Outdoors” page in the Recipe Book shows exactly how to do this, so we don’t need to go into it here. The solution begins with some code that looks something like this:

```
The Great Outdoors is a region. The Forest Path and By the Wall are in the Great Outdoors.
After deciding the scope of the player when the player is in the Great Outdoors:
        repeat with the way running through directions:
                let next-room be the room the way from the location;
                if the next-room is a room:
                        place the next-room in scope.
```

More detail would be needed to produce a convincing illusion of the great outdoors, but this should give you an idea of what’s required: Objects in adjacent rooms have now been added to scope, which means the parser will recognize them when the player refers to them. They can now be examined (but nothing else) from adjacent rooms in the region. For a more complete implementation, with which the player can throw things in a direction and have them sail away out of sight, see “Indoors & Outdoors” in Appendix B.

Enterable Scenery

When the “room” is an outdoor location and a building is nearby, making the exterior of the building scenery is a good idea. This is not hard to do, but it can lead to two small problems that we need to look at.

First, we can’t give the building’s scenery exterior the same name as the room that is its interior. This might easily happen with a one-room building such as a barn or hut. This won’t work:

```
The Farmyard is a room. "A muddy farmyard. A little red barn stands to the north."

The little red barn is scenery in the Farmyard. The description is "The barn is freshly painted a cheerful
bright red."

Little Red Barn is north of the Farmyard. "In the barn you can see some stalls and some hay." [Error!]
```

The compiler will object, quite rightly, that we’ve tried to name two things — a scenery object and then a room — using the same name. Here’s how to solve the problem:

```
The barn-exterior is scenery in the Farmyard. The description is "The barn is freshly painted a cheerful bright red." Understand "little", "red", and "barn" as the barn-exterior. The printed name of the barn-exterior is "little red barn".
```

Now the scenery object has its own name, barn-exterior. But we've given it "little", "red", and "barn" as vocabulary, and we've also told Inform that if the game ever needs to print out its name, it should call it "little red barn."

The second problem we need to deal with is that the player may try ENTER BARN as a command. Unless we say otherwise, Inform will report to the player that scenery objects can't be entered. When the scenery object is a building exterior, this is both rude and misleading. But it's easy to fix. While we're at it, we'll allow the player to use IN as a command, when in the Farmyard, in order to enter the little red barn:

```
Instead of entering the barn-exterior:
	try going north.
Instead of going inside in the Farmyard:
	try going north.
```

If the enterable thing is small (a refrigerator carton or phone booth, for instance) we might want to make it an enterable container rather than a separate room. (See the section on "Enterable Containers & Supporters" in Chapter 3.) With a larger object, making it a separate room will work better.

Doors

In the real world, most indoor rooms are separated by doors. In Inform, a door is a special type of object because it's in two rooms at once and connects the two. Inform understands doors in a basic way, but we'll look at a couple of ways to improve on the default doors.

To create a door, create the rooms on both sides of the door first, but *don't* connect them by mentioning directions. If you mention the directional connection between the rooms, Inform won't let you create a door between them. After creating the rooms, create the door, and tell Inform about its directions with respect to the two rooms:

```
The Entry Hall is a room.
The Billiard Room is a room.
The oak door is a door. The oak door is north of the Billiard Room and south of the Entry Hall.
```

We can't just call the oak door a door: We have to add that extra sentence telling Inform that the door is a door. The following will compile, but it's an error:

```
The Entry Hall is a room.
The Billiard Room is a room.
The oak door is north of the Billiard Room and south of the Entry Hall. [Error!]
```

If we write it this way, Inform will think “oak door” is the name of a room. It will create a third room called oak door, which will lie between the other two rooms. I’m hoping that in a future version of Inform, calling something a door will make it a door, but for now, you have to write the extra sentence.

If you include the code shown above in your game, you’ll soon notice two things. First, the door itself will be mentioned by the game’s output along with any other objects in the room. In other words, a door is not scenery unless we make it scenery. Second, and rather more annoying, if you try to go south from the Entry Hall (or north from the Library), you’ll be told, “You can’t, since the oak door is in the way.” Forcing the player to type OPEN OAK DOOR as a separate command is clumsy, especially since, by default, Inform doesn’t even say that the problem is the door being closed; it only says the door is “in the way.”

Here’s a workaround using a Before rule:

```
Before going through the oak door:
        if the oak door is closed:
                say "(first opening the oak door)[paragraph break]";
                now the oak door is open.
```

Because this Before rule ends without an outcome (that is, it neither succeeds nor fails, as discussed in the Documentation on **p. 12.2**, “How actions are processed,” and **p. 18.10**, “Success and failure”), the action will continue. After your Before rule opens the door, the player will be able to travel through it. Another solution, if you have a lot of doors in your game, is not to make them doors at all. When you need to mention a connection between rooms in your room description, just call it a “doorway.” Most players won’t mind if you sacrifice a bit of realism in order to make the game more friendly.

Two good Extensions are available that address this problem: Locksmith by Emily Short and Implicit Actions by Eric Eve.

You can use the door kind in Inform to make other room-connecting objects, such as gates, ladders, and bridges. A ladder or bridge probably wouldn’t be openable: You’d want it to be permanently open. The reason to make it a door is so that it will be visible in both of the rooms that it connects. For instance:

```
The wooden bridge is a scenery door. The wooden bridge is south of the Meadow and north of the
Forest Path. The wooden bridge is open and not openable. The description is "It's a handsome old
bridge. Curiously, there's no creek running beneath it."
```

It’s handy that Inform understands the command CROSS BRIDGE. You don’t need to code it yourself. But if you create a ladder using the door kind, you’ll need to write an Instead rule to handle CLIMB LADDER:

```
Instead of climbing the rickety ladder:
        try entering the rickety ladder.
```

Locked Doors

A locked door is a slightly different matter. Personally, I feel that a locked door and hidden key don’t make for a very entertaining puzzle. Hundreds of games have included locked doors, and if there’s a doormat or a potted plant anywhere nearby, players will instantly know to LOOK UNDER MAT and SEARCH POT. There’s just not much fun in it anymore. I have played with this cliché in a couple of ways. In one game I included both a locked door that can never be opened because there isn’t a key, and a locked door to which another character spontaneously gives the player a key without even being asked for it. Doors that can only be unlocked from one side (after you find a secret entrance to the room) are slightly more interesting.

Here’s how to create a locked door in Inform:

```
The oak door is north of the Billiard Room and south of the Entry Hall. The oak door is a door. The oak door is scenery. The oak door is lockable and locked. The brass key unlocks the oak door.

Before going through the oak door:
    if the oak door is closed:
        if the oak door is not locked:
            say "(first opening the oak door)[paragraph break]";
            now the oak door is open;
        otherwise if the player carries the brass key:
            say "(first unlocking the oak door with the brass key, then opening the door)[paragraph break]";
            now the oak door is unlocked;
            now the oak door is open.

The player carries the brass key.
```

Several new elements are introduced above. First, if something can be opened and closed, we’re allowed to make it lockable. Lockable is a property of certain kinds of objects: Doors and containers can be made lockable, but nothing else. (Technically, this is not quite true. We can also make a new kind of object — perhaps a detonator — and allow objects of that kind to be lockable and locked.) Next, if something is lockable, at the beginning of the game is can be either locked or unlocked. During the game, the player who has the right key can unlock it — or your code can do so in response to some action by the player. For instance, a door with an old-fashioned bar might be locked and unlocked using the commands BAR DOOR and UNBAR DOOR. Such a door might not have a key at all.

You could write code so that a lockable door became not lockable, but this would only become useful if there was a way within the game to break the lock. Most often, a thing that is lockable will remain lockable (and unlockable, obviously) throughout the game.

If you read **p. 3.13**, “Locks and Keys,” in the Documentation, you’ll learn three different ways to tell Inform that a certain key can be used to unlock a lockable thing. The sentence used above, “The brass key unlocks the oak door,” seems the simplest.

In the current version of Inform, a lockable thing can have only one key. This is not usually a big problem. If you need to, though, you can write an Instead rule that will allow a second key to do the job. Here’s a rather silly example. We’ve already told Inform that the tiny key unlocks the gold amulet, but we also want to be able to use the banana as a key:

```
Instead of unlocking the gold amulet with the banana:
        if the player does not carry the banana:
                say "You don't seem to have the banana.";
        otherwise if the amulet is not locked:
                say "The gold amulet doesn't seem to be locked.";
        otherwise:
                now the amulet is not locked;
                say "You unlock the gold amulet with the banana."
```

If you create the oak door as shown on the previous page, the oak door will automatically be unlocked and then opened if the player carries the key. The only downside of this code is that it assumes the player *knows* the brass key unlocks the oak door. If you want to force the player to discover that fact, you’ll have to work out a way to track what the player knows. Keeping track of the player’s (or the player character’s) knowledge is not too difficult to manage, but the details may differ from one game to the next, so I’ll leave this as an exercise for the advanced Inform programmer.

Secret Doors

Even better than a locked door is a secret door — something that doesn’t appear to be a door at all until its presence is revealed. Secret Doors by Andrew Owen is a simple Extension that allows you to create secret doors. If you include this Extension and then create a secret door, the door will pretend not to exist until something happens in the game that causes it to be revealed. For instance, if the door is disguised as oak wall paneling, this would work:

```
Include Secret Doors by Andrew Owen.

The Billiard Room is a room. "Hand-rubbed oak paneling adds a warm glow above the broad green felt surface of the billiard table."

The Small Windowless Room is a room. "It smells dusty in here, as if the secret door hasn't been opened in ages."

The oak door is north of the Billiard Room and south of the Small Windowless Room. The oak door is a secret door.

The oak wall paneling is scenery in the Billiard Room. The description is "Richly carved oak paneling covers the north wall[if the oak door is open]. One of the panels has been opened; it’s actually a door[otherwise if the oak door is revealed]. One of the panels has an unusually wide seam around it[end if]." Understand "carved" and "panel" as the paneling.

After examining the oak wall paneling for the first time:
        now the oak door is revealed;
        say "One of the panels has an unusually wide seam around it. On closer inspection, the panel
```

```
proves to be a door!"
```

This produces exactly the type of interaction we'd expect of a secret door:

```
>n
You can't go that way.

>open door
You can't see any such thing.

>x paneling
Richly carved oak paneling covers the north wall.

One of the panels has an unusually wide seam around it. On closer
inspection, the panel proves to be a door!

>open door
You open the oak door.

>n

Small Windowless Room
It smells dusty in here, as if the secret door hasn't been opened in ages.
```

Dangerous Doors

One of my students recently asked how to create a door that would slam shut behind the player. I thought this might make an interesting puzzle, so I wrote it up. The example below shows how to use an After rule to affect what happens when the player travels from one room to another:

```
The Corridor is a room. "The corridor stretches east and west from here. A massive stone door stands invitingly open to the north."

The Dank Cell is a room. "This cramped chamber smells of mold, and other things that are a lot less pleasant than mold. You can hear rats scurrying behind the walls. A massive stone door to the south is the only visible exit ... using the term 'exit' loosely, as the door [one of]has just slammed shut with a sound suggestive of finality and doom[or]is firmly shut[stopping]."

The massive stone door is a door. The massive stone door is scenery. The massive stone door is north of the Corridor and south of the Dank Cell. The description is "It's quite an imposing-looking stone door." The massive stone door is open, lockable, and locked.

Instead of closing the massive stone door:
	if the massive stone door is open:
		say "...but it looks so inviting! Why not just step through it and see what's on the other side?";
	otherwise:
		say "It seems already to have closed itself without your lifting a finger."

After going north from the Corridor:
	now the massive stone door is closed;
	say "As you step through the stone door, it swings shut with a terrible loud BOOM!";
	continue the action.
```

You’ll notice that the massive stone door is initially open — but also initially locked. Inform is happy to create a locked door that’s open. As soon as the door is closed, its locked condition will keep it from being opened again.

To learn more about the syntax in the room description, which includes “[one of]” and “[stopping]”, see “Text Insertions” on p. 212.

If this door were used in a real game, there would of course be some sort of hidden exit from the Dank Cell. Devising a hidden exit from a cell that seems not to have any exits … well, let’s just say I’ve seen a few authors try it, and some of their attempts were more convincing than others. The idea that there might be a trapdoor under the straw on the floor doesn’t quite work for me: Why would any sensible jailer ever build a cell with a trapdoor in the floor? Giving the player a tool with which to loosen the bars on the window might be a better puzzle.

Travel by Fiat

Normally, the player moves from room to room under his own steam, by typing GO NORTH (or simply N) and so on. Once in a while, you may want to create a puzzle in which the player will be magically whisked from one room to another by taking some other action. This is easy to do. For example:

```
Instead of rubbing the magic lamp:
	say "You are engulfed in a cloud of sweet-smelling pink smoke! When you emerge, coughing, you find that your surroundings have changed.";
	now the player is in the Harem.
```

One detail should be noted, however: In version 5Z71, if the player is in a dark room and is moved to a lighted room by that type of direct method, a bug in Inform will cause the room description to be printed twice. This bug has been reported, and should be fixed in the next release.

Windows

Real rooms often have windows. Windows have some interesting features. When you look through a window, you’ll normally see what’s on the other side. If the window is open, you may also be able to climb through it. (Or not.) A window is normally in two rooms, like a door. And if the room on one side of a window is lighted, it’s very unlikely that the room on the other side will be a dark room. (For more on dark rooms, see p. 68.)

The easiest way to make a window that the player can actually climb through is to make it a door. This will handle the player’s travel automatically, and will also keep the two sides of the window “in sync” with respect to whether they're open or closed. Since Inform doesn't understand "climb through", we'll also create a new action to let the player use the window:

```
The Lab is a room. "Welcome to the Test Lab. Many devious tests are conducted here. There's a wide window in the north wall."

The Porch is a room. "There's a wide window in the south wall."

The wide window is a door. The wide window is scenery. The wide window is north of the Lab and south of the Porch.

Climbing through is an action applying to one thing. Understand "climb through [something]", "climb in [something]", "climb out [something]", and "climb out of [something]" as climbing through.

Check climbing through:
        if the noun is not a door:
                say "You can't climb through [a noun]!" instead;
        otherwise:
                try entering the noun instead.
```

The commands LOOK THROUGH WINDOW and LOOK IN WINDOW cause Inform to run the SEARCH action. However, LOOK OUT WINDOW and LOOK OUT OF WINDOW aren't understood. We can use an Instead rule to write a description of whatever is on the far side of the window, and add an Understand directive that will give the player two more ways to look through the window.

```
Instead of searching the wide window:
        if the location is the Porch:
                say "There seems to be a laboratory in there, but you can't make out any details.";
        otherwise:
                say "Through the window you can see the porch, but the view isn't good enough for you to make out any details."

Understand "look out of [something]" and "look out [something]" as searching.
```

If we wanted to mention any large objects that would naturally be visible through the window, such as a rocket on a launching pad, we can create them as distant scenery, using the techniques shown earlier in this chapter.

"You can't go that way."

A player who tries, in a certain room, to go in a direction for which there is no map connection in that room, will be told, "You can't go that way." This message has one advantage: It's perfectly clear. The player knows not to bother any more trying to go that direction in that room. But it's not very descriptive. Also, in an open outdoor setting such as a field, being told "You can't go that way" is unrealistic and rather silly. Fortunately, it's easy to write more interesting replacement messages:

```
Instead of going nowhere from Forest Path:
	say "You take a few steps into the forest, but deciding it might not be safe, you return to the path."
```

Note that this rule says “going nowhere *from*”. Writing it as “going nowhere *in*” won’t work. However, there are other situations in which “going in” is needed, and “going from” won’t work. The basic concept is, “going from” assumes that the action of going has succeeded. In the code above, the player has succeeded … in going nowhere.

This code is fine as far as it goes, but the player will get the same output in response to UP or DOWN, which is not so sensible. We might customize our “You can’t go that way” messages further like this:

```
Instead of going up in Forest Path:
	say "The trees have no branches low enough for you to reach them."

Instead of going down in Forest Path:
	say "There are no gaping holes or open mineshafts in the vicinity."
```

Depending on the features of the rooms in your map, you may want to write custom “You can’t go that way” messages that are different for each room. But we’ll borrow an idea from the next section, on Regions, to suggest a more streamlined approach. After telling Inform that Forest Path, Canyon View, and Haunted Grove are all in a region called Forest Area, we can write Instead rules that will apply across the entire region:

```
Instead of going nowhere from the Forest Area:
	say "You take a few steps into the forest, but deciding it might not be safe, you return to the path."

Instead of going up in the Forest Area:
	say "The trees have no branches low enough for you to reach them."

Instead of going down in the Forest Area:
	say "There are no gaping holes or open mineshafts in the vicinity."
```

If you’re using the Secret Doors Extension, you’ll have to do a bit of extra work, because a secret door that hasn’t been revealed will produce the default “You can’t go that way” message, *not* the custom message you’ve written using an Instead rule like those above. But writing a custom message can be used to give the player an in-the-game clue. Following the code about the secret door in the Doors section, above, you could add something like this:

```
Before going through the oak door when the oak door is unrevealed:
	say "You bump your nose on the oak paneling. Odd -- you had somehow absent-mindedly thought there ought to be a door there.";
	rule succeeds.
```

This message will be output if the player tries to go north through the secret door before it’s revealed.

Duplicate Exits & Twisty Connections

Let’s suppose we’ve created a room that’s described like this:

```
The Cellar is down from the Kitchen. "The low ceiling of this little room is festooned with cobwebs, and the floor is dirt. A rough doorway leads east, and the stairs up to the kitchen are built into the north wall."
```

Because we’ve said the Cellar is down from the Kitchen, Inform creates an up/down map connection between the two rooms. But you’ll notice that the room description (in the interest of adding detail) mentions that the stairs are to the north. If the player should try to go north, though, she’ll be told, “You can’t go that way.” This is not too friendly, but it’s easy to fix:

```
Instead of going north in the Cellar, try going up.
```

By the way, Inform insists that we write this as “try going up”, not simply as “go up”. The reason is because there’s no guarantee that the action will succeed. Going up might not work for some reason: Maybe the stairs will collapse, trapping the player in the cellar! That’s why the word “try” is used so often in Inform code.

Sometimes we need to make a one-way connection between rooms. Possibly there’s a chute in the cellar, which the player can go down, but won’t be able to climb back up. It would be friendly to the player to mention that the chute looks a bit treacherous. (This example is similar to one of the first puzzles in Zork, by the way.) There are two ways to do this in Inform.

```
The Cellar is down from the Kitchen. "The low ceiling of this little room is festooned with cobwebs, and the floor is dirt. A rough doorway leads east, and the stairs up to the kitchen are built into the north wall. In the southeast corner is a hole that looks wide enough to enter, but if you climb down it, there’s no guarantee you’ll be able to get back up."

The Spider-Infested Cavern is down from the Cellar. "Spiders! Thousands of them!" Up from the Spider-Infested Cavern is nowhere.
```

Here we’ve created a one-way map connection by telling Inform that the up direction from the cavern leads nowhere. The other way to do it would be with an Instead rule:

```
The Spider-Infested Cavern is down from the Cellar. "Spiders! Thousands of them!"

Instead of going up in the Spider-Infested Cavern, say "The hole is too steep for you to climb back up."
```

The advantage of using an Instead rule is that you can tell the player exactly what the travel problem is.

In the original game of Adventure, some of the rooms were connected not in the normal way, but with “twisty” connections. One of the main puzzles in the game involved figuring out how to draw a reliable map. This type of puzzle is not used much in modern games, but once in a while you may want to create a connection between two rooms that is a bit twisty rather than straight. For instance, the room description might tell the player, “You can go east around the corner of the building.” To return after going east, you might need to go north. Here’s how to create this type of connection:

```
Deeper in the Cellar is east of the Cellar. "This little room smells awful."
The Cellar is north of Deeper in the Cellar.
West of Deeper in the Cellar is nowhere.
South of the Cellar is nowhere.
```

Notice that we have to tell Inform that after going east from the Cellar, we can’t get back where we started by going west — we have to go north. That’s why the two “nowhere” lines have been added. This makes the model world a little more realistic. It’s also a good idea if you’re including Exit Lister by Eric Eve. This Extension will list the exits from every room in the status line — and if a room with only one actual exit shows two exits on the status line, the player may get a little confused.

For a more concise syntax that will produce “dog-leg” connections between rooms, see Example 7 (“Port Royal 2”) in the Documentation. We can do it this way:

```
East of the Cellar is north of Deeper in the Cellar.
```

An even more twisty and confusing map connection is to have one of the exits of a room lead back into the same room. To do that, you would write something like this:

```
East of Deeper in the Cellar is Deeper in the Cellar.
```

Changing the Map During the Game

In most of the games you may write, the model world will work the way you want it to if you create connections between rooms at the beginning of the game (that is, when creating the rooms in your source) and then leave the connections alone. But once in a while, you may want a connection between rooms to disappear while the game is being played — maybe because the rooms are in a cave complex and there has been a cave-in blocking a tunnel. Or maybe a magic door has suddenly appeared in a room, creating a connection where there wasn’t one before. The magic door might even have a mind of its own, and wander off again. (Inform doesn’t allow doors to be moved around during the game; if you want a magic door to move around, you’ll have to write some extra code to create an ordinary object that responds to the player’s commands as if it were a door.)

To get rid of an exit from a room, change the exit to nowhere. (It’s usually a good idea to do this to both sides of the connection between rooms, just in case there’s another route the player can use to get around to the other side of the blocked exit.) To restore an exit or create a new one, change the exit so that it points to the room that is the destination.

Here’s a simple test game that shows how to do it. In this game, if the player tries the command UP before the Tree House has been visited, the only result will be a nudge (“You gaze speculatively....”) Once the tree has been climbed, however, the UP command will work.

```
The Forest Path is a room. "Tall old trees surround you. That old oak tree, for instance."
The Tree House is a room. "What an amazing view!"

The old oak tree is scenery in the Forest Path. The description is "It's a sturdy old tree."

Instead of going up in the Forest Path:
    if the Tree House is unvisited:
        say "You gaze speculatively at the old oak tree. It looks as if it might be climbable.";
    otherwise:
        continue the action.

Instead of climbing the old oak tree:
    change the up exit of the Forest Path to the Tree House;
    change the down exit of the Tree House to the Forest Path;
    say "You clamber up the tree....";
    now the player is in the Tree House.
```

Next we’ll look at a somewhat artificial example that has one or two added features. In this short game there are two north-south connections (between the Living Room and the Kitchen, and between the Bathroom and the Bedroom), but only one of them will exist at any time, depending on which button you push. To make the changes easier to test, I’ve put the buttons on a backdrop, so that they’re always present no matter what room you’re in. I also included Exit Lister, which will display the current exits in the status line. Notice that the status line will be updated each time you press a button.

```
Include Exit Lister by Eric Eve.

The Living Room is a room.
The Kitchen is south of the Living Room.
The Bedroom is east of the Living Room.
The Bathroom is east of the Kitchen.

The floating button holder is a backdrop. It is everywhere. It is not scenery.

Rule for writing a paragraph about the floating button holder:
    say "A button holder is floating in mid-air here. On it are a red button and a green button."

The red button is part of the floating button holder. The green button is part of the floating button holder.

Instead of pushing the red button:
    change the south exit of the Living Room to nowhere;
    change the north exit of the Kitchen to nowhere;
    change the south exit of the Bedroom to the Bathroom;
    change the north exit of the Bathroom to the Bedroom;
    say "Living Room south is gone, Bedroom south is open."

Instead of pushing the green button:
    change the south exit of the Living Room to the Kitchen;
    change the north exit of the Kitchen to the Living Room;
    change the south exit of the Bedroom to nowhere;
    change the north exit of the Bathroom to nowhere;
```

```
	say "Living Room south is open, Bedroom south is gone."
```

This neat little trick won’t work with doors, unfortunately. Inform has rigid ideas about how doors work. It’s possible to make a door seem to disappear during the game, even though it’s still present. Doing this is awkward and error-prone, but the place to start would be with the Secret Doors Extension, which is discussed later in this chapter. A better approach might be, as mentioned earlier, to create an ordinary thing that responds to the user’s commands as if it were a door.

Regions

As explained on **p. 3.4** of the Documentation (“Regions and the index map”), using regions is a nice way to keep a larger map organized. After you’ve defined some regions, you’ll also be able to use some of Inform’s world-building features more easily. This is because you can test what region the player is in, and do something with the information. For instance, after creating a region called Forest, you could do this:

```
Every turn when in the Forest:
	say "[one of]A bird chirps.[or]You hear a soft rustling in the bushes.[or]A butterfly flits past you.[at random]".
```

(The elements inserted in square brackets in a double-quoted text allow Inform to produce a text output that changes. For more on this topic, see “Text Insertions” in Chapter 8 of the *Handbook*.) Creating a region and adding rooms to it is simple. You do it like this:

```
Forest is a region. Forest Path, Haunted Grove, and Canyon View are in Forest.
```

I’ve found that it’s safer to define a region *after* creating the rooms that will be in it — that is, the region definition should be below the rooms in the source code. This is because the room object is created the first time it’s mentioned in the code. Inform understands that the only things that can be in regions are rooms, so it will add a new room when you first create the region, if it doesn’t already know about that room. Some of the ways that you can write sentences that create rooms will confuse Inform if it already knows about a room that has that same name. Also, you need to be sure to use exactly the same room name in the region list that you use when creating the room. If there’s a typo in the room name in the region list, Inform will cheerfully create a second, featureless room whose name has a typo in it. This can lead to hard-to-find bugs.

For consistency, and to make your code easier to read, I suggest always using the word “Area” in the names of your regions. So I would edit the code above to read like this:

```
Forest Area is a region. Forest Path, Haunted Grove, and Canyon View are in Forest Area.
```

You can create one region that’s entirely within another region, but Inform won’t let two regions overlap one another. When creating a region that’s contained in a larger region, it’s important to mention the rooms in only one region definition. The following won’t work:

```
Room1 is a room. Room2 is a room. Room3 is a room. Room4 is a room. Room5 is a room.
```

```
The Big Area is a region. Room1, Room2, Room3, Room4, and Room5 are in the Big Area.

The Little Area is a region. Room1, Room2, and Room3 are in the Little Area. [Error!]
```

Here’s how to get the desired result:

```
Room1 is a room. Room2 is a room. Room3 is a room. Room4 is a room. Room5 is a room.

The Little Area is a region. Room1, Room2, and Room3 are in the Little Area.

The Big Area is a region. The Little Area, Room4, and Room5 are in the Big Area.
```

Notice that the Little Area is defined before the Big Area. This is so we can refer to the Little Area in the region definition for the Big Area. If we try to do it in the other order, Inform will think “Little Area” is a room — because anything in a region has to be a room. It will then get confused when we tell it that Little Area is a region. If you do it as shown above, it will work.

If you need to create regions (that is, groups of rooms) that overlap or that change during the course of the game, the workaround is fairly simple: Don’t use regions at all. Instead, use properties. For example:

```
A room can be cursed or uncursed. A room is usually uncursed.

Every turn when the player is in a cursed room:
        say "Oooh, scary!"
```

At any point in the game, you can change a room from cursed to uncursed or vice-versa with a single line of code:

```
now the Dormitory is uncursed.
```

Backdrops

Rooms in Inform are not built out of anything — they’re just empty, featureless containers that you can move objects into and out of. In particular, you might expect that a room would have a ceiling, walls, and a floor. But if you try X CEILING, X FLOOR, or X WALL in your game, you’ll be told, “You can’t see any such thing.” Likewise, in an outdoor setting, X SKY and X GROUND won’t work. Most players will probably understand this convention, and won’t even think to try interacting with the walls, ceiling, and floor. But if you mention a wall, ceiling, floor, sky, or ground in the room description, creating it as scenery would be a good thing to do.

Scenery is always in one specific room. Inform also provides a special kind of scenery called a *backdrop.* A backdrop object is unusual because it can be in several places at once.

An easy way to add a little realism to your game is to use backdrops to create sky, ground, ceiling, floor, and so on only where they’re needed:

```
The sky is a backdrop. "A clear and cloudless blue." The sky is in the Forest Area.
The ground is a backdrop. "It's rather dirty.” Understand "dirt" as the ground. The ground is in the Forest Area.
```

Now if the player types X SKY in any room in the Forest Area, the game will reply, “A clear and cloudless blue.” That’s a definite improvement! (Note that “The description is” is not required for the descriptions of backdrops.) As we add indoor rooms, perhaps in a castle, they won’t be in the Forest Area region, so a player who tries X SKY while in the castle will be told, very appropriately, “You can’t see any such thing.”

Backdrops are more versatile than you might expect. A backdrop could be used, for instance, to create a river that’s present in several rooms. In one game I wanted a windowsill (a supporter — see p. 86) that could be reached from both inside and outside the room. At Emily Short’s suggestion, I created the window as a backdrop, so that it could be in two rooms at once, and then made the windowsill a part of the window.

Removing a Backdrop

Getting rid of a backdrop entirely during play is easy. If we’ve created, for instance, some thick fog, when the player does something to cause the wind to blow, we can write:

```
remove the fog from play;
```

Removing a backdrop from certain rooms while leaving it in other rooms is slightly tricky, however. Inform has no command for this, but we can create a routine that will do it. What we need to do is give all of the rooms where the backdrop is to be found a property. Since we’re going to create some fog, we’ll allow rooms to be foggy or not foggy. We’ll also give the player a giant bellows. Pumping the bellows will dispel the fog, but only from the current room.

```
Include Plurality by Emily Short. [This is so we can write a more streamlined Check rule for our new action.]

A room can be foggy or not foggy. A room is usually not foggy.

The Desolate Moor is a room. "A gloomy treeless waste stretches out on all sides[if the fog is in the Moor], but you can't see very far, because the fog is closing in[otherwise]. You can see a path that extends north and south from here[end if]." The Desolate Moor is foggy.

The Haunted Grove is north of the Desolate Moor. "Thin, widely spaced trees of a mournful character surround you[if the fog is in the Grove]. It's difficult to see where you might go from here, because the fog presses close among the trees[otherwise]. A path leads south out of the grove[end if]." The Haunted Grove is foggy.

The Bog is south of the Desolate Moor. "The ground here is quite moist[if the fog is in the Bog], and the fog is thicker[otherwise]. A path extends out of the bog to the north[end if]." The Bog is foggy.

The thick gray fog is a backdrop. The description is "Tendrils of gray fog drift across the land."
When play begins:
	move the fog backdrop to all foggy rooms.

The player carries some giant bellows. The indefinite article of the giant bellows is "a".

Pumping is an action applying to one thing. Understand "pump [something]" as pumping.
Check pumping:
	say "[The noun] [is-are] not something you can pump."

Instead of pumping the giant bellows:
	if the location is foggy:
		say "As you pump the bellows with great vigor, the fog blows away!";
	otherwise:
		say "You've already dispelled the fog here.";
	now the location is not foggy;
	update backdrop positions.
```

The important thing in this example is the Instead rule. It makes the location not foggy, and then updates the backdrop positions (see **p. 8.8** in the Documentation, “Moving backdrops”).

Dark Rooms

The very first interactive fiction, a game called Adventure, was set in a cave. Like most caves, it was dark. As long as the player was carrying a lantern, the game could be played in what we now consider the normal way. But any room where there wasn’t a light source would be pitch-dark.

In a dark room the player can’t examine anything, won’t be able to see things lying on the floor, won’t be able to LOOK to read the room description, most likely can’t read what’s written in books and on pieces of paper, and may not be able to find an exit reliably so as to go back to a lighted room. You should still be able to take inventory in a dark room, and (depending on how the game is written) you may be able to touch things.

Today, the rooms in most games have light, so no lantern is needed. Rooms in Inform are always lighted unless you state otherwise.

Page 3.15 in the Documentation, “Light and darkness,” explains how to create dark rooms, and how to create objects (such as torches and flashlights) that are lit. It’s a convention of IF that dark rooms are completely dark, but Example 322, “Zorn of Zorna,” shows how to create a system in which the amount of light in a room can be dim.

A dark room to which you have to bring a light source is a standard type of puzzle, and most players will have no trouble figuring out what they need to do. You might consider some variation that makes it harder to bring a light source. Two obvious ideas are a flashlight for which you have to find batteries and a candle whose flame blows out due to gusts of wind. Both of these have been used in a lot of games, though. To create a really good dark room puzzle, you'll need to come up with something more imaginative — perhaps a glass jar full of fireflies, a secret lever that opens a louvered skylight, or a friendly ghost who glows in the dark and can be persuaded to follow you.

To create an object in Inform that provides light, all you need to do is say that it does:

```
now the torch is lit;
```

In a real game, you might want the torch to burn out after a fixed number of moves, making a timed puzzle. Or finding the matches with which to light the torch might be a separate puzzle. If the object is lit from the start of the game, you'd do it this way:

```
The ghost is a person. The ghost is lit.
```

But what if the player is in a dark room, and you want her to be able to find the switch so as to turn on the light? This situation can trip up the novice author, and here's why: Inform assumes that when things are in a dark room, the player can't sense or manipulate them. If the light switch is in the dark, the player will never be able to use it.

Here's a simple example that shows how to get around this problem. It involves using a special rule to add the light switch to scope. Also included are customized rules about how to announce darkness and — much more important for this example — how to print the description of a dark room. These methods are described in Chapter 17 of the Documentation, on **pages 17.17** ("Printing the announcement of darkness") and **17.20** ("Printing the description of a dark room"). We need a special way to print the description of the Library when it's dark, because otherwise the player will have no idea that there's a light switch in the room. "Guess what you can do in a dark room" is a very unfair type of puzzle.

```
The Library is a room. "Thousands of books line the shelves. An old-fashioned lamp with a green lampshade sits on the table. Mounted on the wall beside the door is a light switch." The Library is dark.

The oak table is a supporter in the Library. The old lamp is on the table.

The light-switch is a device in the Library. The light-switch is scenery. Understand "light" and "switch" as the light-switch.

Instead of switching on the light-switch:
	now the lamp is lit;
	say "By groping on the wall near the door, you find a light switch. Click -- the lamp goes on."

Instead of switching off the light-switch:
	now the lamp is not lit.

Rule for printing the announcement of darkness:
	if the location is the Library:
		say "Click -- the room is plunged into inky darkness." instead;
	otherwise:
		say "It is now pitch dark in here!"
```

```
Instead of switching off the old lamp:
	try switching off the light-switch.

After deciding the scope of the player when the location is the Library:
	place the light-switch in scope.

Library darkness count is a number that varies. Library darkness count is 0.

Rule for printing the description of a dark room:
	if the location is the Library and the library darkness count is greater than 0:
		say "You can't see a thing -- but as you grope around in the darkness, your fingers touch a switch on the wall.";
	otherwise:
		increase the library darkness count by 1;
		say "It is pitch dark, and you can't see a thing."

Test me with "look / switch switch".
```

If you look closely at the code above, you'll see a couple of other refinements as well. The game starts in the Library (since there's only one room), and we don't want the message about groping around and finding the light switch to appear at the very beginning of the game. So we create a counter (a variable) called "library darkness count." At the beginning of the game, this is 0 — but the first time the description of a dark room is printed, the counter is increased to 1. Thereafter, when the player uses the LOOK command in the Library, she'll discover the light switch.

I also used an Instead rule so that if the player tries SWITCH OFF LAMP, the command will be redirected to SWITCH OFF LIGHT-SWITCH. This keeps all of the handling of the lamp's on/off condition in one place — in the switch. The switch is an Inform **device** (see p. 116), which as explained in Chapter 3 is one of the kinds of object that Inform allows you to create.

Backdrops in Dark Rooms

Inform assumes, by default, that a backdrop is something that a player will want to look at. For that reason, backdrops are not present in dark rooms, even when you've written code that states the backdrop is everywhere. If your game includes a backdrop that the player may want to smell, you'll need to do a little extra work. Here's how to create a backdrop that can be smelled even in a dark room:

```
The Forest is a room. "All around you stretches the Great Northern Forest. To the east is a cave."

The Cave is east of the Forest. The Cave is a dark room.

The air is a backdrop. It is everywhere. The description of the air is "You can't see the air!"

Instead of smelling the air:
	if the player is in the Cave:
		say "Dank and foul.";
	otherwise:
		say "Fresh."
```

```
After deciding the scope of the player when in darkness:
	place the air in scope.
```

The “After deciding the scope of the player” rule insures that the air will be present in the Cave.

Seemingly Identical Rooms

Once in a while, you might want to design an area of the map in which there are several rooms that have the same name and description (such as “You are in a maze of twisty little passages, all different.”). Internally, in your code, each of your rooms has to have a unique name of its own. But it’s easy enough to give several rooms the same printed name and description:

```
Land1 is a room. "The landscape stretches out to the horizon on all sides." The printed name of Land1 is "Landscape".

Land2 is south of Land1. "The landscape stretches out to the horizon on all sides." The printed name of Land2 is "Landscape".

Land3 is west of Land1. "The landscape stretches out to the horizon on all sides." The printed name of Land3 is "Landscape".
```

...and so on. To the player, all of these rooms will initially appear to be identical. But as in the original game of Adventure, from which the line about twisty little passages is borrowed, the player can easily drop a different object in each room and then draw a map based on the locations of the various objects. When I find myself (against my better judgment) designing a puzzle of this sort, I usually try to come up with a plausible reason why the player character wouldn’t want to drop anything, such as “Better not. You might never be able to find it again.” This at least forces the player to come up with a fresh approach to mapping the region. It’s not a very original puzzle, but you might want to try something of the sort, so it’s worth mentioning.

Floorless Rooms

As mentioned at the beginning of this chapter, when the player uses the DROP command to get rid of objects that are being carried, the convention of IF is that the objects end up on the floor. This is usually a sensible way for objects to behave — but what if the “room” is the top of a tree?

Creating a floorless room turns out to be very easy:

```
A floorless room is a kind of room. A floorless room has a room called the drop-location.

The Forest Path is a room. "Tall old trees surround you. That magnificent oak, for instance."

The old oak tree is scenery in Forest Path. "Plenty of low-hanging branches. You could probably climb it."
```

```
Instead of climbing the old oak tree:
	try going up.

Top of the Oak Tree is up from the Forest Path. "The view from here is inspiring!" Top of the Oak Tree is a floorless room. The drop-location of Top of the Oak Tree is Forest Path.

After dropping something in a floorless room (called F):
	move the noun to the drop-location of F;
	say "[The noun] drop[s] out of sight."
```

The only thing you need to be careful of, with this code, is that every floorless room *must* have its drop-location defined. If you forget to do this, things that are dropped will end up (oddly enough) in the starting location of the game — the first room defined in your source.

When Is a Room "Visited"?

In Chapter 1 we introduced the idea of verbose and brief mode. When a game is in verbose mode, the text of each room's description property is printed each time the player enters the room. In brief mode, however, the description is printed only the first time the player enters a room. Thereafter, the description is printed only if the player uses the LOOK command. (Note that other aspects of the room's full description, namely a list of the non-scenery objects that are in the room, will always be printed when the player enters a visited room, even if the game is in brief mode.)

At the beginning of the game, only the room where the player character starts the game is marked as visited. All other rooms are unvisited. What's interesting is that a room is marked as visited only *after* its description property is printed for the first time. This fact allows us to change the room description after the first time the player reads it:

```
The Throne Room is north of the Entry Hall. "This high-vaulted chamber is grandiose in the extreme. [if unvisited]Your first sight of its magnificence quite takes your breath away. [end if]Each of the dozens of richly woven tapestries lining the walls surely cost more than all of the shabby hovels and mangy livestock in the humble village where you were born."
```

Here, the sentence about the view taking your breath away will appear only the first time the Throne Room is seen.

If the game is in superbrief mode, the description property won't print even on the player's first visit. This mode is not used by most players, because room descriptions generally contain useful information! But it's important to understand that you, the author, don't get to control whether the player will see the wonderful room descriptions you've written.

Or rather … there's a way to control it, though it would be rude. Let's suppose that something happens in a room in the course of the game that changes a room so drastically that you'd like to change both its description and its printed name. Perhaps there has been an explosion in the kitchen. The actual name of the room object won't change — it's still called Kitchen in the source code. But now we want it to be shown in the game's output as "Ruins of the Kitchen". This is easy. As part of the code that produces the explosion (whatever that might be), we write this:

```
now the printed name of the Kitchen is "Ruins of the Kitchen";
now the description of the Kitchen is "About five tons of icky cake batter have splattered everywhere!"
```

The existence of the BRIEF and SUPERBRIEF commands, however, means that you can't be certain your players will ever read about the cake batter. As part of your explosion in the kitchen, then, you could include, "now the Kitchen is unvisited;". But if the player has cavalierly switched to superbrief mode, that won't work: They still won't see the new description. Arguably, this is their problem, not yours. But if you find it an insupportable intrusion on your authorial prerogatives, you can get rid of brief and superbrief modes entirely, forcing the player to see the complete room description each time a room is entered. Here's how to do it:

```
Use full-length room descriptions.
Understand the command "brief" as something new.
Understand the command "superbrief" as something new.

Mode-change-refusing is an action out of world, applying to nothing. Understand "brief" and
"superbrief" as mode-change-refusing.

Carry out mode-change-refusing:
        say "Sorry -- this game is always in verbose mode."
```

More About Room Descriptions

We've already seen (in the section on "Backdrops," earlier in this chapter) a few examples of rooms whose description can change. These changes were produced by using square brackets to insert bits of code in the room description itself. This can be done with an if-test, as shown above in the example of the Throne Room, or with a [one of] insertion. There are other possibilities as well. These techniques are discussed in more detail in the section on "Text Insertions" in Chapter 9 of the *Handbook*, "Phrasing & Punctuation," but here's an example of each technique to get you started:

```
The Bog is south of the Desolate Moor. "The ground here is quite moist[if the fog is in the Bog], and
the fog is thicker[otherwise]. A path extends out of the bog to the north[end if]."

The Sunlit Clearing is east of the Bog. "Sunlight streams down into this charming and cheerful space.
[one of]Butterflies flit about you[or]A chipmunk scampers past[or]Songbirds twitter in the trees[at
random]."
```

There's more to room descriptions than this, however. Each time the player enters a new room or uses the LOOK action, Inform assembles a text to print out. The room's name is the first part of this text; it's followed by description you've written for the room. After printing out your room description, Inform will mention anything in the room that it considers interesting — mainly any visible objects that are not scenery.

The rules that are used for assembling the rest of the text are called the carry out looking rules. They're complex, and this *Handbook* is not the place to dissect them line by line. Instead, we'll mention a few of the things that you may want to do to customize the appearance of your game. You may often want to control how various objects are listed as part of the complete room description. A large chunk of Chapter 17 of the Documentation, “Activities,” is devoted to this complex and useful topic, from 17.9, “Deciding the concealed possessions of something,” to 17.26, “Printing a locale paragraph about.” Close study of these pages and the Examples that go with them will give you plenty of ideas to experiment with.

Let’s look at a couple of ways to control the way objects are mentioned. We’ll start with this code:

```
Use the serial comma.

The Living Room is a room. "Your comfy living room."

The paper clip is in the Living Room.

Buffy the Labrador retriever is a female animal in the Living Room.

The sugar candy doll house is in the Living Room.
```

The output reads like this:

```
Living Room
Your comfy living room.

You can see a paper clip, Buffy the Labrador retriever, and a sugar candy
doll house here.
```

Notice that the list of items that Inform has constructed is in a separate paragraph. Also, notice that the items are listed in exactly the order they were mentioned in the source code. (The statement, “Use the serial comma,” puts a comma after the next-to-last item in the list.)

Now we’ll add one sentence to the code for Buffy. This sentence will be an **initial appearance** (explained more fully in Chapter 3). This new sentence, beginning “Buffy the Labrador retriever is lounging...”, is not a description. Remember, only when we’re creating rooms and scenery can we write a description that doesn’t start with the phrase “The description is”. So Buffy has, as yet, no description; she only has an initial appearance:

```
Buffy the Labrador retriever is a female animal in the Living Room. "Buffy the Labrador retriever is
lounging here, shedding hair all over the place."
```

The initial description will remove Buffy from the list, and give her her own paragraph in the output. Now the output reads like this:

```
Living Room
Your comfy living room.

Buffy the Labrador retriever is lounging here, shedding hair all over the
place.
```

```
You can also see a paper clip and a sugar candy doll house here.
```

Notice the word “also” in the last sentence. Inform has noticed that there was a separate paragraph about Buffy before the basic list, so it added an “also”.

Next, we’ll add a scenery supporter to the Living Room. (Containers and supporters are discussed in Chapter 3, “Things.”) Since it’s scenery, it won’t be mentioned as part of the list of objects in the complete room description, so we’ll want to add a bit to the room description property itself:

```
The Living Room is a room. "Your comfy living room, complete with a falling-apart coffee table from Particle Board Heaven."

The coffee table is a scenery supporter in the Living Room. The description is "You really ought to replace it with something more upscale."
```

Adding the coffee table will have no effect on the way objects are listed in the room. But if we make one more change, we’ll see something new in the output. We’re going to put the paper clip on the coffee table:

```
The paper clip is on the coffee table.
```

This changes the output significantly:

```
Living Room
Your comfy living room, complete with a falling-apart coffee table from
Particle Board Heaven.

On the coffee table is a paper clip.

Buffy the Labrador retriever is lounging here, shedding hair all over the
place.

You can also see a sugar candy doll house here.
```

This is Inform’s normal handling: A scenery supporter will be mentioned in its own paragraph, *but only if something is on it.* However, this is not true of scenery containers. The contents of a container are presumed to be less immediately visible than what’s on a supporter, so they won’t be mentioned unless the player actually looks in (searches) the container.

Mentioning the paper clip as shown above is probably just fine — there’s nothing very remarkable about a paper clip. But what if the object on the table is the priceless Etruscan figurine Mandy has been raving to you about throughout the first half of the game? In that case, this would be dull:

```
On the coffee table is an Etruscan figurine.
```

In a post to the newsgroup, Eric Eve suggested a simple way to get a more dramatic output, using the printing a locale paragraph about activity (see **p. 17.26** of the Documentation, “Printing a locale paragraph about”).

```
Rule for printing a locale paragraph about the coffee table when the Etruscan figurine is on the coffee table and the number of things on the coffee table is 1:
	now the Etruscan figurine is mentioned;
	say "On the coffee table you espy the priceless Etruscan figurine Mandy has been raving about!";
	continue the activity.
```

Or perhaps you want that dramatic sentence to print only the first time the player enters the room rather than every time the figurine is still standing in solitary splendor on the coffee table. In that case, create a truth state (a true-false variable) that will shut off the special output after its first occurrence:

```
Figurine-mentioned is a truth state that varies. Figurine-mentioned is false.
Rule for printing a locale paragraph about the coffee table when the Etruscan figurine is on the coffee table and figurine-mentioned is false:
	now the Etruscan figurine is mentioned;
	now figurine-mentioned is true;
	say "On the coffee table you espy the priceless Etruscan figurine Mandy has been raving about!";
	continue the activity.
```

Chapter 3: Things

A lot of the fun of interactive fiction comes from being able to pick up and use objects in the model world. At the beginning of this chapter, you'll learn how to create things. Once you've started creating things, though, you'll find that leaving them lying around where players can just pick them up won't give players much of a challenge. We'll also look at ways to hide things in the model world, so that finding a useful object becomes a puzzle.

Most of the things you'll create in a game will have uses. Some kinds of things (such as keys) already have uses that Inform understands. But if you create a bicycle pump so that the player can blow up a flat tire, you'll also need to create an *action*, as described in Chapter 4 of the *Handbook,* so that the player can use the command INFLATE TIRE WITH PUMP. Creating actions so that things become useful is an important part of writing a good game.

Pretty much every object you create in your Inform game is a thing. Computer science majors would tell you that "thing" is a base class — a kind, in Inform's terms — and that many other kinds, such as person and door, *inherit* from the thing kind. But we don't need to worry about that jargon. In this *Handbook*, we'll usually call an object in your game a thing only when it isn't any other kind of thing, such as a door or container.

Creating Things

The most basic way to create a new thing is to tell Inform that it exists:

```
The paintbrush is a thing.
```

This is enough to create an object called the paintbrush. But as yet, the paintbrush is not anywhere in the model world. Sometimes this is what you want: You may need to create objects that are offstage at the beginning of the game, so that your code can move them onstage later. More often, we'll write an assertion that will cause the thing to appear in a certain place at the beginning of the game. We can do it this way:

```
The Tool Shed is a room.
The paintbrush is in the Tool Shed.
```

Another way to do exactly the same thing is this:

```
The Tool Shed is a room.
The paintbrush is here.
```

I suggest not using "is here," however. If you need to move paragraphs or whole sections of your code around for any reason, or if you forget how you've defined the paintbrush and add more code after the Tool Shed assertion and before the sentence that creates the paintbrush, bad things can happen (that is, your code may contain bugs). It's safer always to tell Inform specifically where a thing is to be placed at the beginning of the game. This is a safe way to create things, however:

```
The player carries a bowling ball.
The duchess wears a diamond tiara.
```

Inform will understand that both the bowling ball and the diamond tiara are new things that need to be added to the model world. It will also understand that the diamond tiara is **wearable** — that it's something characters can put on and take off.

Containers and supporters will be discussed in detail later in this chapter. Briefly, a supporter is a thing that acts like a table, so objects can be placed on it. This would also work well:

```
The Tool Shed is a room.
The workbench is a supporter in the Tool Shed.
The screwdriver is on the workbench.
```

Now Inform knows that the screwdriver is a thing, and knows exactly where to put it at the start of the game.

The most important thing about things is that the player can pick them up and carry them around. This is not true of some other types of objects: The player won't be allowed to pick up a door or a person, for example.

Inform knows almost nothing else about newly created things, though. It doesn't know the difference between a screwdriver and a bowling ball. In order to have these two things behave in different ways in your game, you'll have to write some code. (To learn how to do this, see Chapter 4 of the *Handbook*.)

Things vs. Kinds

New Inform writers sometimes get into trouble by using the word "kind" when it's not needed. This word has a special meaning in Inform. It refers to types of objects — what traditional computer programmers call "classes." The usual reason to create kinds is so that you can write code that will apply to several different objects, like this:

```
A fruit is a kind of thing. The banana is a fruit. The orange is a fruit. The plum is a fruit. The apple is a
```

```
fruit.

Before eating a fruit:
	say "You’re not hungry right now." instead.
```

Here’s an example, adapted from a recent post on the newsgroup rec.arts.int-fiction, of how you can get in trouble using the word “kind” when you don’t need to:

```
A phaser is a kind of thing.
A phaser can be either set to stun or set to kill.
Instead of examining the phaser, say "Your shiny Mark Five phaser. Dual setting - stun and kill. [if the phaser is set to kill]It is currently set to kill.[Otherwise]It is currently set to stun."
A phaser is usually set to stun.

The player is carrying a phaser. Understand "Mark", "Five", and "shiny" as the phaser.

[Error: will not compile!]
Instead of setting the phaser to "kill":
	now the phaser is set to kill;
	say "You set your phaser to kill."
```

The problem is that the Instead rule is attempting to change the setting not of any individual phaser but of the entire *kind.* Inform won’t let the author do that. There are a couple of ways to solve the problem. We can rewrite the Instead rule so that Inform knows which phaser to set:

```
Instead of setting a phaser (called P) to "kill":
	now P is set to kill;
	say "You set your phaser to kill."
```

Here’s another way to do exactly the same thing without using the “called” phrase to create a temporary name for the object:

```
Instead of setting a phaser to "kill":
	now the noun is set to kill;
	say "You set your phaser to kill."
```

Noun & Second Noun

Notice the term “the noun” in the code above. The words “noun” and “second noun” have a special meaning in Inform. The noun is a temporary value (a variable) that refers to whatever object the player mentioned in the most recent command. For instance, if the player types EAT THE PIE,” the pie object becomes (temporarily) “the noun.” Likewise, the second noun is the second object in a two-object command, such as STAB TROLL WITH SWORD. Here, the troll is the noun and the sword is the second noun. You should always use “noun” and “second noun” in code when you’re not sure what object the player will be referring to. For more on variables, see p. 224.

Both Instead rules for the phaser will work. Either way, Inform now understands that it's supposed to set one particular phaser to kill, and it's happy to do so. In general, creating a kind and then an object that has the same name as the kind is best avoided. Sometimes it's necessary; for instance, the door kind is built into Inform, and it would be silly to try to avoid calling a door a door. But if you can come up with a specific word for your new kinds, a word that is not used by any of the objects, you'll create fewer confusing bugs.

Unless you're planning to have several phasers in your game, though, there's no reason to make phaser a kind in the first place. Instead, make it a thing:

```
The player carries a phaser. The phaser can be set to kill or set to stun. The phaser is set to stun. The description is "Your shiny Mark Five phaser. Dual setting - stun and kill. [if the phaser is set to kill]It is currently set to kill.[Otherwise]It is currently set to stun." Understand "Mark", "Five", and "shiny" as the phaser.

Instead of setting the phaser to "kill":
	now the phaser is set to kill;
	say "You set your phaser to kill.
```

Now the Instead rule for setting the phaser works the way the original author of this code intended.

One reason for the confusion here, by the way, is that Inform quite often ignores "a" and "the" when compiling. There are a few situations where it notices "a" or "the", such as when constructing lists, but in this particular case it pays no attention whatever to the difference that most of us would see between "a phaser" and "the phaser".

The Names of Things

When creating objects, it's a good idea to add an adjective or two to the name. When possible, the adjective should be unique, not a word that can also be used with other objects. This is not required, but failing to do it can get you into trouble. Here's a simple example that shows why:

```
The red potion vial is an open container on the table.
The red potion is in the red potion vial. [Error!]
```

This code won't compile. The compiler complains, "You wrote 'The red potion is in the red potion vial': but this asks to put something inside itself, like saying 'the bottle is in the bottle'." The compiler gets confused because all of the words that can refer to the red potion can also apply to the vial. The solution is simple: Just give the vial and potion their own names:

```
The red vial is an open container on the table.
The healing potion is in the red vial.
```

In a real game, we'd need to write more code than this. For one thing, the player can pick up the potion (which would probably be a liquid). This example is just about naming things.

Here's a more complicated version of the same problem. Let's suppose you've created three keys in your game — a rusty key, a silver key, and a third object called simply the key. Inform will let you do this, as long as you define the plain old key (not an object called "the plain old key" but the key with no adjectives) first in your source code. During the course of the game, the player might be carrying all three of the keys, and might need to unlock a door using the one that you've called simply "key". This will cause a bug to appear in your game. The output will look like this:

```
>unlock door with key
Which key do you mean, the rusty key, the silver key, or the key?

>key
Which key do you mean, the rusty key, the silver key, or the key?
```

The question above goes by the fancy term *disambiguation*. The parser is attempting to figure out what the player's input means. It comes up with two or more possible meanings, and has no way to decide which is correct, because the input is ambiguous — the parser doesn't know which key object the player is referring to. The command UNLOCK DOOR WITH KEY could mean several different things, so the parser needs to ask the player to provide more information. The parser tries to get rid of the ambiguity by asking the player to add some detail.

As long as the player wants to use the silver key or the rusty key, this is not a problem: The parser's question can be answered SILVER or RUSTY and the game will proceed as planned. But if the player needs to use the plain key, the one for which there's no adjective, the player is stuck: There's no way to tell the parser that you mean the plain old key, other than by going into a different room, entering the commands DROP SILVER KEY and DROP RUSTY KEY, and then returning to the room with the locked door. Inform provides an advanced technique for getting around this problem — but why create the problem for yourself in the first place, when it's so much easier to give each key its own adjective when naming it?

Inform is very unusual among programming languages, by the way, in allowing you to name objects using spaces between words. Most programming languages would require that the various key objects above be named silver_key and rusty_key (using the underscore character), silverKey and rustyKey, or with some other combination of letters.

Long Names

Normally, Inform looks only at the first nine letters in each word. The rest of the letters are ignored. This is true both for the names of things in your code and for the words in commands that the player types. Normally nine letters are plenty. (In the very first text-based games, only the first five or six letters in the player's commands were read, and that wasn't enough.) But as **p. 3.1** ("Descriptions") of the Documentation points out, if you happen to put a superhero and a superheroine in the same room, the player will quite likely get the wrong result from the command KISS SUPERHEROINE.

There are two ways to get around this problem, if you ever need to. The easy way requires compiling to Glulx. If you're not already compiling your game to Glulx, go to the Settings page in the Inform IDE and click the Glulx button. Then add the following line near the top of your source:

```
Use DICT_WORD_SIZE of 12.
```

Now the player will be able to use both SUPERHERO and SUPERHEROINE in commands, and Inform will be able to tell the difference. You can use as large a number as you'd like, but if your game is written in English, it's hard to see how you would ever need more than the first 12 letters of each word.

If you need to compile to the Z-machine standard (possibly because you want your game to be playable on handheld devices), you won't be able to change the value of DICT_WORD_SIZE, so you'll need to resort to a little trickery. The next example is a stripped-down version of some code I used in my game "A Flustered Duck." Some leprechauns are having a picnic in a meadow, and there's also a single leprechaun (with whom the player can converse) wandering around playing a fiddle. The trick is, we're going to use words shorter than nine letters, but allow both the player's input and the game's output to use the longer words.

```
After reading a command:
	let N be indexed text;
	let N be the player's command;
	replace the regular expression "leprec" in N with "lpc";
	change the text of the player's command to N.

The Grassy Knoll is a room. "Some leprechauns are having a picnic here. One leprechaun is sauntering around playing a sprightly tune on a fiddle."

The lpchauns is scenery in the Grassy Knoll. The description is "They're enjoying their picnic." The printed name of the lpchauns is "leprechauns".

The lpchaun is a man in the Grassy Knoll. The description is "He's playing a fiddle." The printed name of the lpchaun is "leprechaun".
```

The first block of code strips a few letters out of the player's commands. If the player types X LEPRECHAUN, the game will see the command as X LPCHAUN. Then we use Inform's handy printed name property so that the object whose real name is lpchaun will be displayed as "leprechaun", and similarly for the lpchauns.

Initial Appearance

When you create a new object and put it in a room, it will be mentioned right after the room description, but in a very basic way. If we've created a banjo, for instance, the room description will end with, "You can see a banjo here."

Inform objects have a special property called **initial appearance**. If an object has an initial appearance, this will be used in the room description until the object has been picked up by the player.

If Inform sees a quoted sentence just after a new object has been created, it will know that this is the initial appearance of the object. We could create our banjo like this:

```
The Meadow is a room. "Wildflowers carpet the meadow."

The old banjo is in the Meadow. "A banjo lies forgotten among the wildflowers." The description is "It's a 1938 Selmer 5-string."
```

The sentence “A banjo lies forgotten among the wildflowers” is the initial appearance of the banjo. The term “initial appearance” is actually the name of an Inform property. Properties are a type of data that’s attached to an object. The description of an object is another of its properties. When we give the banjo an initial appearance, this is what will happen when the player enters the Meadow:

```
Meadow
Wildflowers carpet the meadow.

A banjo lies forgotten among the wildflowers.
```

Having the line about the banjo off in a paragraph by itself looks a little odd, but that’s only because the room description of the meadow is so brief. If the room description were three lines long, having a new paragraph about the banjo would look perfectly natural.

But as one of my students discovered, Inform is a little fussy about what it thinks quoted sentences refer to. This code may look the same to a human reader, but it’s an error:

```
There is an old banjo in the Meadow. "A banjo lies forgotten among the wildflowers." The description is "It's a 1938 Selmer 5-string." [Error!]
```

Do you see the difference? I replaced “The old banjo is in...” with “There is an old banjo in....” Because this sentence begins with “There is,” Inform now thinks the sentence “A banjo lies forgotten...” is a description of the Meadow, which already has a description. The same thing will happen if we leave out the initial appearance of the banjo:

```
There is an old banjo in the Meadow. The description is "It's a 1938 Selmer 5-string." [Error!]
```

Now Inform thinks “It’s a 1938 Selmer 5-string” is a second description of the Meadow, again because we used the sentence form “There is....” The moral of the story is, avoid creating objects by saying, “There is....” It’s legal to do so, but if you want to do it, you have to do it like this:

```
There is an old banjo in the Meadow. The initial appearance of the banjo is "A banjo lies forgotten among the wildflowers." The description of the banjo is "It's a 1938 Selmer 5-string."
```

Now Inform can figure out what those sentences refer to (the banjo), so the game will compile and run the way we want it to.

An initial appearance will be used only until an object is picked up for the first time by the player character. There may be times when you’d like an object to be referred to in a non-default way in room descriptions on a continuing basis, or possibly in several non-default ways depending on which room it’s in. The following code, which I’ve adapted slightly, was suggested by Al in a newsgroup post:

```
Rule for writing a paragraph about the shovel: say "[if the shovel is in the Garage]Your shovel lies in the corner against the wall.[otherwise if the shovel is in the Tool Shed]On a shelf is your handy shovel.[otherwise if the shovel is in the Work Site]Your shovel is stuck in the ground here.[otherwise]You seem to have left the shovel lying here.[end if]".
```

Adding Vocabulary Words

When you create an object, such as the paintbrush we created earlier in this chapter, Inform is smart enough to understand two things at once. First: you, the author, can refer to the object in your code as a paintbrush, and the compiler will know what you mean. Second: the player who is playing your game can also call it a paintbrush, and the parser will know what the player means. When you first start learning Inform, these two facts may seem to be almost the same — the object is a paintbrush, and that's what it's called. What could be complicated about that? But in reality, the name of an object for internal purposes (that is, when you're writing code) is not at all the same thing as the word(s) the player can use to refer to the object. They're often the same, but they don't have to be. In fact, Inform allows you to make them completely different if you need to. (See the section on "The Names of Things" in Chapter 8 of the *Handbook*.)

After creating an object, you'll almost always want to add extra vocabulary words to it — words that the player can use to refer to it. With a paintbrush, for instance, the player will quite often want to call it a brush. But the parser won't understand that word unless you tell it to:

```
The paintbrush is in the Tool Shed. Understand "brush" as the paintbrush.
```

Once you've added the second sentence, the player can use the word "brush" to refer to the paintbrush — but the author can't. The author can only refer to an object using the actual word or words that are used in the sentence that creates the object.

When you write a description for a new object, you'll quite often find that you're adding extra adjectives. These should always be added as vocabulary:

```
The paintbrush is in the Tool Shed. The description is "The bristles of the paintbrush are stiff with dried paint." Understand "brush", "bristles", "stiff", "dried", and "paint" as the paintbrush.
```

If you also have a can of paint in your game (which wouldn't be surprising if you've created a paintbrush), the word "paint" will end up being ambiguous. It will be able to refer either to the brush or to the paint can (and possibly to the paint *in* the can as well), and also to the command PAINT. Handling all of the possibilities in a case like this can get a little tricky. We're not going to go through the whole troubleshooting process here — it's just something to be aware of when you start creating objects whose names and vocabulary words overlap.

Inform will attempt to keep track of what you mean when writing the game, and the parser will try to figure out what the player means when entering the word PAINT. If the parser can't figure out what the player meant, it will ask questions. You can help the parser by writing "does the player mean" rules (see **p. 16.18** of the Documentation, "Does the player mean...").

The main idea to keep in mind is this: Always give your own objects in the code individual names. Add an adjective to the name so that it will be unique: Never call one object “paint can” and another object just “paint” (or “can”), because that can make it difficult for the parser to clarify the situation. Here’s a transcript that illustrates what can happen if you slip up on this point:

```
>take paint
Which do you mean, the paint, or the paint can?

>paint
Which do you mean, the paint, or the paint can?
```

It has become impossible to refer to the paint object itself, as long as the paint can object is in scope. There are ways to work around this problem, but the best practice is, avoid creating the problem in the first place.

Conditional Vocabulary

Most of the objects in a game will likely have the same vocabulary words from one end of the game to the other. But there are situations in which you may want to switch certain words on or off. For instance, if the vase gets broken during the game, the player should be able to refer to it as BROKEN VASE — but not otherwise.

The simplest way to do this is to refer to properties of the object when listing the vocabulary words:

```
The porcelain vase is here. The vase can be broken or unbroken. Understand "broken" and
"shattered" as the vase when the vase is broken.

Instead of attacking the vase:
	now the vase is broken;
	now the printed name of the vase is "broken porcelain vase";
	say "You shatter the vase."
```

Notice that the Instead rule both changes the property of the vase and changes the vase’s printed name.

For more complex story situations, you may want to create scenes (as described in Chapter 8 of the *Handbook*) and make the vocabulary words that the player is allowed to use depend on whether a scene is happening:

```
Daytime is a recurring scene. Daytime begins when the sun is part of the sky. Daytime ends when the
sun is not part of the sky.

Understand "twittering" and "chirping" as the birds when daytime is happening.
```

For more on giving objects sets of words (like broken and unbroken) as properties, see the section on “Word Properties” later in this chapter.

Containers & Supporters

Every object in your model world (except backdrops and doors, which operate in a slightly different way) is either in a room, or it's off-stage. If an object is off-stage, it's nowhere, at least at the moment. But even when something is off-stage, it's still part of the game, and could be moved into play later on. (See the section "Moving Things Around," below, to learn how to do this.) The question of where things are located is rather interesting, as we'll discover in the section "Testing Where a Thing Is." Before we dig into that question, we need to introduce two new kinds: containers and supporters. These were introduced briefly at the end of Chapter 2, in the section "Room Descriptions." Now it's time to take a closer look.

A container is, as you can probably guess, an object that can contain other things. That is, the player can put things *in* a container. Most of the time, if you need a basket or a cupboard in your game, you'd make the basket or cupboard a container.

A supporter is like a table: it's an object that you can put things *on*. Inform understands the difference between a supporter and a container. If the player tries to put something *in* a table, she'll be told, "That can't contain things." If she tries to put something *on* a container, such as a cupboard, she'll be told, "Putting things on the cupboard would achieve nothing."

By default, a supporter is assumed to be a piece of furniture: It's not scenery (unless you make it scenery), but it's fixed in place. If you want to create a portable supporter, such as a tea tray, you need to tell Inform that it is not fixed in place:

```
The tea tray is a supporter on the buffet. The tea tray is not fixed in place.
```

There are ways to create an object (such as a chest of drawers) that the player can put things either in or on; see the section "Objects that Have Parts," later in this chapter. Such an object can behave like a supporter in response to some player commands, and behave like a container at other times. For the rest of the discussion in the section you're reading now, though, we're going to assume that containers and supporters are entirely different. The main thing they have in common is that the player can put movable things on or in them.

Containers have some special properties that are not available for supporters. A container can be **openable**. If you create a container but don't tell Inform that it's openable, Inform will assume that it's permanently open — that it operates pretty much like a basket. On the other hand, if you tell Inform that your cupboard is openable, then the commands OPEN CUPBOARD and CLOSE CUPBOARD will work just the way the player would expect them to (though the cupboard won't have an actual door — you can give it a door, but that's a more complex coding challenge; for details, see "Objects that Have Parts," later in this chapter).

Your game will automatically keep track of whether each container is opened or closed. If it's closed, the player won't be able to see or take anything that's inside. On the other hand, if the container is not only openable but **transparent**, the player will be able to see what's inside even when the container is closed, but won't be able to take what's inside. The transparent property is good for creating things like bird cages and glass-front sideboards.

A container that is openable can start the game either **open** or **closed**. Inform understands that "not open" means the same thing as "closed." So we could create a basic cupboard like this:

```
The cupboard is a closed openable container in the Kitchen.
```

If a container is openable, it can also be lockable. If it's lockable, it can start the game either locked or unlocked. And as **p. 3.13** of the Documentation ("Locks and keys") explains, things that are lockable can be given keys.

In fact, Inform is a little pickier than this: By default, you can only say that something is lockable if it's a container or a door. If you want to create a small gold locket as a piece of jewelry, and give it a key, one easy way to do it would be to make it a container — after which you'll probably want to write an Instead rule to prevent the player from putting anything at all into it (unless putting a small photograph or a magic bean into the locket is the solution to a puzzle). This is not difficult to do:

```
The player carries a small gold locket. The locket is an openable container. The locket is lockable and
locked. The tiny gold key unlocks the locket.

The player carries a bowling ball and the tiny gold key.

Instead of inserting something into the small gold locket:
        if the locket is closed:
                say "You'd need to open the locket to do that.";
        otherwise if the player does not carry the noun:
                say "You're not holding [the noun].";
        otherwise:
                say "There's not room for [the noun] in the locket."

Test me with "put ball in locket / unlock locket with key / open locket / put ball in locket".
```

Note the use of "inserting something into" in the code above. One of the common mistakes authors make is trying to write a rule for "putting something in" something else. Inform lets the player use the syntax PUT BOWLING BALL IN LOCKET, but the action that this produces is inserting it into, not putting it in.

But there's an easier way. By adding a little more code, we can make the locket lockable and openable even though it's not a container. The reasons why this type of code works the way it does are a bit technical, so we won't worry about them here. What you need to know is that before you make something openable or lockable (unless it's a container or a door), you have to tell Inform that it *can be* openable or lockable. Here's how:

```
The player carries a locket. The locket can be lockable. The locket is lockable. The locket can be
openable. The locket is openable. The locket can be open. The locket can be locked. The locket is not
```

```
open. The locket is locked.

The player carries the tiny gold key. The tiny gold key unlocks the locket.
```

If you do this, you’ll find that the locket can be opened and closed, locked and unlocked — but it can’t contain things, because it isn’t a container. The code above doesn’t include a description of the locket (which should probably change depending on whether it is open or closed). Nor does the code tell the player what will be discovered on opening the locket, which would presumably be important information.

But we were talking, a couple of pages back, about a kitchen cupboard. Let’s make the cupboard a little fancier:

```
The glass-front cupboard is an openable transparent lockable container in the Kitchen. The cupboard is closed and locked. The brass key unlocks the cupboard.
```

In order to allow the player to UNLOCK THE CUPBOARD DOOR, we might want to give the cupboard an actual glass door. To do that, see “Objects that Have Parts.” Another way would be to make the word “door” a synonym for the cupboard.

Inform assumes that the cupboard is *permanently* openable, and that it can be opened or closed during the course of the game. But your code can change a container to not openable during the course of the game. This is something you will rarely need to do, but someday you might want to create an openable lockable container that is closed and locked, and that has *no* key. If the solution of the puzzle is HIT CUPBOARD WITH HAMMER, you might create a new action called attacking it with and then write some code along these lines:

```
Carry out attacking the cupboard with the hammer:
        now the cupboard is open;
        now the cupboard is not openable;
        now the cupboard is unlocked;
        now the cupboard is not lockable;
        say "You smash the cupboard door with the hammer, and it springs open."
```

Once this cupboard has been smashed, it’s no longer lockable and no longer openable.

Inform has a standard way of describing containers and supporters and their contents, but we can change this if we need to. To look what usually happens (unless we write some new code), let’s create a simple game that has three containers and a supporter. One of the containers (the cupboard) and the supporter (the table) are scenery. Another container, the suitcase, is closed and openable. The final container, a basket, is not scenery and not openable. That is, the basket is permanently open. Here’s the code for the game:

```
The Living Room is a room. "Your standard American living room, equipped with a table and a cupboard."

The table is a scenery supporter in the Living Room.
The cupboard is a closed openable scenery container in the Living Room.

The basket is a container in the Living Room.

The suitcase is a closed openable container in the Living Room.

The pear is on the table. The apple is in the cupboard. The plum is in the basket. The carrot is in the suitcase. The player carries a banana.

Test me with "open suitcase / look / take carrot / look / open cupboard / look / take plum / look / take pear / look"
```

If you haven't used the TEST ME command, this would be a good time to try it out. Create a new, empty project, paste the code shown above into it, and click the Go! button. When the game appears in the right-hand page, enter the command TEST ME. Inform will run through all of the commands in the "Test me with" line, above. Here's the output:

```
Living Room
Your standard American living room, equipped with a table and a cupboard.

On the table is a pear.

You can also see a basket (in which is a plum) and a suitcase (closed)
here.

>test me
(Testing.)

>[1] open suitcase
You open the suitcase, revealing a carrot.

>[2] look
Living Room
Your standard American living room, equipped with a table and a cupboard.

On the table is a pear.

You can also see a basket (in which is a plum) and a suitcase (in which is
a carrot) here.

>[3] take carrot
Taken.

>[4] look
Living Room
Your standard American living room, equipped with a table and a cupboard.

On the table is a pear.

You can also see a basket (in which is a plum) and a suitcase (empty) here.
```

```
>[5] open cupboard
You open the cupboard, revealing an apple.

>[6] look
Living Room
Your standard American living room, equipped with a table and a cupboard.

On the table is a pear.

You can also see a basket (in which is a plum) and a suitcase (empty) here.

>[7] take plum
Taken.

>[8] look
Living Room
Your standard American living room, equipped with a table and a cupboard.

On the table is a pear.

You can also see a basket (empty) and a suitcase (empty) here.

>[9] take pear
Taken.

>[10] look
Living Room
Your standard American living room, equipped with a table and a cupboard.

You can see a basket (empty) and a suitcase (empty) here.
```

This output shows several things about how Inform handles containers and supporters. Sometimes the game will add sentences of its own after the room description that you wrote; other times, it won't add anything. Here are the normal rules (which, again, we can change if we need to):

1) If a container is scenery, it doesn't get its own paragraph of output after the room description — not even when it's open and something is visible inside. If the player wants to see what's in a scenery container, she has to LOOK IN it or SEARCH it. (These two commands lead to the same action.)
2) If a supporter is scenery (as the table is in this game), it gets a separate paragraph after the room description, but only if something is on it. When nothing is on it (after we TAKE PEAR in this game) the table no longer rates a paragraph of its own. Inform assumes that things sitting on supporters are more immediately visible than things in open containers. But if nothing is on the scenery supporter, Inform assumes that it has already been mentioned in the room description, and adds nothing.
3) If a container is not scenery, Inform will normally add a paragraph about it after the room description (assuming it's in the room — not if the player is holding it). If the container is openable and closed, Inform will add "(closed)" after mentioning it. If it's open and contains something, Inform will list the contents. If it's open and empty, Inform will say "(empty)".

Stealthy Containers

The logic shown above is fine as a basic way of designing a game, but you may run into situations in which it doesn't work well. For instance, you might want it to be less than obvious that a closed openable container is actually a container at all. Getting rid of the "(closed)" text would solve that. Here's how:

```
Rule for printing room description details of a closed container: stop.
```

Once the closed container is picked up by the player, however, the "(closed)" will reappear when the player takes inventory. To prevent that, we need another few lines:

```
Rule for printing the name of a closed container (called C) while taking inventory:
	say "[printed name of C]";
	omit contents in listing.
```

The initial appearance property can also help get us get specific printouts for containers, as can the printing a paragraph about activity. Let's look at a more complete example. We have a hollow log (an open container) in which is a gold key. We don't want to make the log scenery, because the player needs to be able to pick it up and take it to the river in order to cross the river.

Our first thought might be to write it this way:

```
The Forest is a room. "Tall old trees stand on all sides."
The hollow log is in the Forest.
The gold key is in the hollow log.
```

Oddly enough, if we tell Inform that the key is in the log, we don't even have to mention that the log is an open container; Inform will figure that out. (For that matter, if we say that Steve is wearing a hat, Inform will figure out that Steve is a person without our needing to say so, because only people can wear things; and also that the hat is wearable.) But when we compile this code, we'll find that the key is in plain sight, so the puzzle will fall flat. Here's the output:

```
Forest
Tall old trees stand on all sides.

You can see a hollow log (in which is a gold key) here.
```

One way to get rid of the mention of the key is to create an initial appearance for the hollow log, like this:

```
The Forest is a room. "Tall old trees stand on all sides."
The hollow log is in the Forest. "A hollow log lies fallen next to the path."
The gold key is in the hollow log.
```

The new sentence, "A hollow log lies fallen next to the path," is *not* the description of the log. (We haven't written a description yet.) It's the log's initial appearance. As explained earlier in this chapter, in the section "Initial Appearance," Inform will use the initial appearance we've given to the log instead of writing its own paragraph about the log. But it will only use the initial appearance until the log has been picked up. If the player picks up the log and drops it again, Inform will toss out the initial appearance and go back to its standard way of mentioning the log, thus revealing the gold key to the player.

A slightly better solution is to give Inform a new rule for writing a paragraph about the hollow log. Here's how to do it:

```
Rule for writing a paragraph about the hollow log:
	say "[if the location is the Forest]A hollow log lies fallen next to the path[otherwise]A hollow log is lying here[end if]."
```

When we've written a "rule for writing a paragraph about", Inform will *always* use this rule when adding the log to a room description rather than construct its own sentence about the log, so the gold key will remain hidden until the player actually thinks to search the log. (Of course, the key might fall out when the player picks up the log. That's a more complex situation, which I'll leave you to work out for yourself. Hint: Try writing an After rule, and include the phrase "for the first time".)

The downside of writing a new "rule for writing a paragraph about" is that if we *do* want the contents of the hollow log to be mentioned at some point in the game, we'll have to write a more complex rule that will tell Inform when we do or don't want this extra output, and how the extra text should be put together.

We're not quite out of the woods yet, though (so to speak). Remember, we didn't make our hollow log scenery, because we want the player to be able to pick it up and move it to another location. While carrying it, the player might think to take inventory. Here's the result:

```
>i
You are carrying:
  a hollow log
    a gold key
```

Oops — the gold key has been revealed again. This happens because the contents of open containers and supporters that the player is carrying are listed when an inventory list is constructed. To prevent this, we need to add another new rule. This is like the one mentioned a couple of pages back, but here we'll apply it specifically to the hollow log rather than to all closed containers (because, of course, the hollow log isn't closed; it's just behaving in a mysterious way because it's a puzzle):

```
Rule for printing the name of the hollow log while taking inventory:
	say "hollow log";
	omit contents in listing.
```

Sometimes we may want an open container to list its contents. We just don't want Inform to print out an annoying reminder that the container is closed or empty every time it includes the container in a list. **Page 17.10** of the Documentation, "Printing the name of something," shows how to handle this. Adapting the code there slightly, we'll create a pillbox. While we're at it, we'll restrict the pillbox so that it can only contain pills. This type of restriction is often useful with small containers. (Another way to prevent the player from putting a bowling ball into the pillbox would be to use the Extension called

Bulk Limiter by Eric Eve.)

```
A pill is a kind of thing. The blue pill is a pill. The red pill is a pill. The yellow pill is a pill.

The player carries a paper clip and a pillbox. The pillbox is a closed openable container. The
description is "The pillbox is small and white.” Understand "box", "white", and "pill box" as the pillbox.
The blue pill, the red pill, and the yellow pill are in the pillbox.

Rule for printing the name of the pillbox while not inserting or removing or opening:
	if the pillbox is open:
		if something is in the pillbox:
			say "pillbox (containing [a list of things in the pillbox])";
		otherwise:
			say "empty pillbox";
	otherwise:
		say "pillbox";
	omit contents in listing.

Instead of inserting something into the pillbox:
	if the pillbox is not open:
		say "You can't put anything into the box until you open it.";
	otherwise if the noun is not a pill:
		say "That won't fit into the pillbox.";
	otherwise:
		continue the action.
```

Notice the line above that says “(containing [a list of things in the pillbox])”. This is a handy bit of syntax; Inform can construct lists during the game if we include code that explains, in a general way, what should be included in the list. Incidentally, this is one of the situations where the compiler will notice the difference between “a” and “the”. If we say “[a list of things...]”, the list will be printed out in the game as “a blue pill, a red pill, and a yellow pill”. But if we say “[the list of things...]”, the list will appear as “the blue pill, the red pill, and the yellow pill”.

Earlier, when we were writing code for the hollow log, we were trying to prevent it from revealing its contents. But sometimes we have to help Inform go the other direction. The contents of open containers will be listed when the container is mentioned in a list that Inform prints out — but the contents won’t be mentioned when an open container is simply EXAMINEd. If an open container is examined, the game won’t bother to list what’s inside. The player needs to LOOK IN or SEARCH the container to learn what’s in it. With an ordinary container like a suitcase, forcing the player to take that extra step is a bit silly. So here’s a suitcase that will list its contents (and its state) when examined:

```
The suitcase is in the Train Station. The suitcase is a closed openable container. The description is "A
vintage item of brown leather luggage.[if open] In the suitcase you can see [a list of things inside the
suitcase].[otherwise] It's closed.[end if]".
```

That works pretty well, as long as there’s something in the suitcase. But if the suitcase is empty, we get the following output:

```
>x suitcase
```

```
A vintage item of brown leather luggage. In the suitcase you can see
nothing.
```

That “you can see nothing” is a bit crude. I’d rather have the game say “The suitcase seems to be empty.” The difficulty is this: Inform won’t let us embed one if-test inside of another in a double-quoted string. We’re not allowed to do this:

```
The description is "A vintage item of brown leather luggage.[if there is something in the suitcase][if
open] In the suitcase you can see [a list of things inside the suitcase].[otherwise] It's closed.[end if]
[otherwise] It seems to be empty.[end if]". [Error!]
```

The error report from the compiler is helpful. It says this: “a second '[if ...]' text substitution occurs inside an existing one, which makes this text too complicated. While a single text can contain more than one '[if ...]', this can only happen if the old if is finished with an '[end if]' or the new one is written '[otherwise if]'. If you need more complicated variety than this allows, the best approach is to define a new text substitution of your own ('To say fiddly details: ...') and then use it in this text by including the '[fiddly details]'.” This tells us exactly how to write code that will do what we want. Here’s one way to do it, using an [otherwise if] construction:

```
The description is "A vintage item of brown leather luggage.[if there is something in the suitcase and
the suitcase is open] In the suitcase you can see [a list of things inside the suitcase].[otherwise if
open] It seems to be empty.[otherwise] It's closed.[end if]".
```

And here’s another way, using Inform’s handy To Say phrase:

```
The suitcase is a closed openable container. The description is "A vintage item of brown leather
luggage. [suitcase-desc details]."

To say suitcase-desc details:
    if the suitcase is open:
        if the number of entries in the list of things inside the suitcase is at least 1:
            say "In the suitcase you can see [a list of things inside the suitcase]";
        otherwise:
            say "It seems to be empty";
    otherwise:
        say "It's closed".
```

Pay close attention to the way the periods at the ends of the sentences are handled. We want the game to print out exactly one blank line after the description of the suitcase, so we put the period just before the closing quote in the *main* description, not in the sentences in the suitcase-desc details. Notice also the space before “ [suitcase-desc details]”. This insures that there will be a space between sentences, no matter which details are printed out.

The same thing happens if the player examines a table — Inform won’t bother to mention what’s on the table unless we tell it we want that information to be included in the description. With a table, though, printing the line “It seems to be empty” in response to an EXAMINE action would be a bit silly, so the code can be simpler:

```
The billiard table is a scenery supporter in the Billiard Room. The description is "The table is big and
green[if there is something on the table]. On the table you can see [a list of things on the table][end
if]."
```

Here’s another way to accomplish the same thing. This one moves the question of whether there’s something on the table into a different block of code.

```
The billiard table is a scenery supporter in the Billiard
Room. The description is "The table is big and
green[list-table-stuff]."

To say list-table-stuff:
    let L be the list of things on the billiard table;
    if the number of entries in L > 0:
              say ". On the table you can see [a list of
things on the billiard table]".
```

Which of these forms you use is purely a matter of taste. I like the second one because it makes the logic easier to see at a glance.

Sneaky Supporters

There may be times when you’d like to force the player to examine a supporter before the game reveals what’s on it. By default, Inform *will* list what’s on a scenery supporter in a room description. If you don’t like this effect, you can override it globally (that is, everywhere in your game) like this:

```
The describe what's on scenery supporters in room descriptions rule is not listed in any rulebook.
```

When you add this line, the player will have to examine scenery supporters in order to see what’s on them — and you’ll have to use the code given a few paragraphs back to insure that the things on all of your supporters will appear in response to an EXAMINE command.

If you want to force the player to examine the billiard table to notice the cue ball that’s lying on it, but you want that effect to apply *only* to the billiard table, not to any other supporter in the game, the way to do it is, first, *not* to make the table scenery, and second, to add a “rule for writing a paragraph about.” Here’s a complete example that does this. Notice that we’re not mentioning the billiard table in the room description of the Billiard Room, because the table is not scenery, so Inform will add a paragraph about it after the room description, using the new rule we’ve added:

```
The Billiard Room is a room. "Comfortable-looking leather chairs stand against the oak-paneled walls
of this room. Overhead, a single hooded light fixture shines down."

The billiard table is a supporter in the Billiard Room. The description is "The billiard table is big and
green[list-table-stuff]."

The white ball and the cue chalk are on the table. The indefinite article of the cue chalk is "a piece of".
```

```
Rule for writing a paragraph about the billiard table:
        say "A handsomely appointed billiard table dominates the center of the room."

To say list-table-stuff:
        let L be the list of things on the billiard table;
        if the number of entries in L > 0:
                say ". On the table you can see [a list of things on the billiard table]".
```

This produces a nice result: The player has to X TABLE to notice the ball and the chalk.

Looking Under & Looking Behind

Experienced IF players know that authors like to hide things under other things — under a bed, for example. When a player enters a room and sees a bed, he's bound to try LOOK UNDER BED before very long. Inform's default response is, "You find nothing of interest." Creating a non-default response is easy, but not too useful for game design:

```
Instead of looking under the bed:
        say "Nothing but dust bunnies."
```

On **p. 6.6** of the Recipe Book you'll find several ideas about how to hide things under other things. If we want to, we can have the player "find" something by moving it from off-stage into the room, or directly into the player's inventory, like this:

```
The odd sock is a thing. The odd sock can be found. The odd sock is not found.

Instead of looking under the bed:
        if the odd sock is found:
                say "There's nothing else of interest under there, just a few dust bunnies.";
        otherwise:
                now the odd sock is found;
                now the player carries the odd sock;
                say "Under the bed you find an odd sock, which you retrieve."
```

This works nicely, up to a point. For your first game, it may be all you want or need. But there are two potential problems lurking here. First, it won't be possible to put anything (including the sock) under the bed:

```
>put sock under bed
I didn't understand that sentence.
```

And second, if your game limits the number of items the player can carry at once, giving the sock to the player automatically (as shown above) may cause the player to be carrying more than the allowed number of things. I've found that this can trip up Inform's process of automatically inserting excess items into the player's holdall (a handy bag for carrying excess inventory).

This is a good reason for moving the sock into the room rather than adding it directly to the player's inventory. But then the player has to TAKE SOCK as an extra command, which is a bit annoying. If the player finds the sock, shouldn't picking it up happen at the same time?

Another way to hide the odd sock under the bed is to include Underside by Eric Eve. This is a handy Extension. Once this Extension has been included in your game, hiding the sock under the bed is easier:

```
The double bed is a supporter in the Bedroom. The double bed is fixed in place. An underside called under#bed is part of the double bed.

The odd sock is in under#bed.
```

The name "under#bed" is not special; it's just a good idea to use a name that the player is not likely to type.

When an object is in an underside, it won't be mentioned in a room description, and it won't be included in the object list if the player tries to TAKE ALL.

There is no Extension for hiding things behind other things. Most often, if you want to do this, you'd be hiding something behind a picture on the wall, or behind a couch, and the way to let the player find it would be with the command TAKE PICTURE or MOVE COUCH. Inform does not include looking behind as an action, though. Here's how to set that up:

```
Looking behind is an action applying to one thing and requiring light. Understand "look behind [something]" as looking behind.

Check looking behind:
        say "You find nothing interesting behind [the noun]."

Instead of looking behind the couch:
        say "It's jammed up against the wall. In order to see what's behind it, if anything, you'll need to pull it out from the wall."
```

Pulling something away from the wall would require another action, as well as a way to test the position of the couch within the room. Is it against the wall, or has it been moved? Let's add that possibility:

```
The couch is in the Living Room. The couch can be moved or not moved. The couch is not moved. The description is "An overstuffed couch stands [if not moved]against the wall[otherwise]in the middle of the room[end if]."
Instead of pushing the couch:
        try pulling the couch.
Instead of pulling the couch:
        if the couch is moved:
                say "It's already out in the middle of the room.";
        otherwise:
                now the couch is moved;
                move the odd sock to the Living Room;
                say "As you wrestle the couch out into the middle of the room, you find an odd sock
```

```
behind it."
```

Allowing the player to push the couch back against the wall would be a bit more complicated. For one thing, you'd have to make sure that the odd sock wouldn't keep popping back into the room each time the couch was moved.

Another way to handle this situation, and without forcing the player to move the couch, would be to open Underside, block-copy the code part of it into your game, and edit the code as needed (quite a lot of editing would be required) to create behind-spaces and the actions looking behind and putting behind. I'll leave this as an exercise for the advanced Inform programmer.

Take All

Since the TAKE ALL command came up in the section on "Looking Under & Looking Behind," we may as well take a quick look at how to deal with it. Experienced IF players have a tendency to type TAKE ALL after entering a new room, just to see what's lying around that isn't nailed down. This is a useful command, and many players feel it's their right to be able to use it. If you disable it entirely, some players may not be happy with your game. By default, though, Inform will try to let the player take even scenery objects (and then report that the scenery objects are not portable) and people (and then report that the person wouldn't care for that). This is all rather silly. An easy way to prevent it is shown on **p. 17.34** ("Deciding whether all includes") of the Documentation:

```
Rule for deciding whether all includes scenery: it does not.
Rule for deciding whether all includes a person: it does not.
```

After ruling out all of the scenery, we might decide to let "all" include a particular scenery item, perhaps because the player's attempt to take it will reveal something interesting:

```
The billiard table is scenery in the Recreation Room.

Instead of taking the billiard table:
	say "It's much too heavy for you to pick up, but when you try to lift it, it rocks slightly, as if the floor beneath it is uneven."

Rule for deciding whether all includes the billiard table: It does.
```

Enterable Containers & Supporters

Some containers and supporters might be big enough that the player could reasonably enter them — a chair or bed, for instance. Here is a simple example that includes both:

```
The Bedroom is a room. "Your basic bedroom. It's equipped with a bed and a chair."

The bed is an enterable scenery supporter in the Bedroom. The chair is an open enterable scenery container in the Bedroom.
```

The player will get able to GET IN BED or SIT ON BED, but not LIE IN BED or LIE DOWN IN BED, because the action "lie in" has not been defined. If you aren't sure how to do that, turn to Chapter 4 of the *Handbook,* "Actions." The standard command used by the player to get out of an enterable container is STAND (or STAND UP, or EXIT). Annoyingly, GET OUT OF BED is not recognized by the parser. In order to handle this command, we need to write a little extra code:

```
Getting out of is an action applying to one thing. Understand "get out of [something]" as getting out of.

Carry out getting out of something:
    try exiting instead.
```

When an openable container is enterable, the player will be allowed to close it from the inside. Inform understands that the inside of a closed container is dark, so if the player enters the container and then closes it, the player will be in darkness — unless carrying a light source, of course.

```
The refrigerator is an openable enterable container in the Kitchen. The description is "An old white
General Electric fridge." Understand "fridge" as the refrigerator. The refrigerator is open.
```

By default, the player will be allowed to pick up things that are in the room (that is, on the floor) even when in or on an enterable container or supporter. This is not too realistic, so you might want to prevent it. I'm a little nervous about the syntax of the next bit of code, because at any given moment either C or S will be nothing ... but it seems to work:

```
Before taking something:
    if the player is enclosed by an enterable container (called C) or the player is enclosed by an
enterable supporter (called S):
        if the noun is not enclosed by C and the noun is not enclosed by S:
            say "You can't reach [the noun] from here." instead.
```

The syntax shown above, in which we use the phrases "(called C)" and "(called S)" to create temporary local values for things so that we can test them, is one that you'll use a lot in writing if-tests in Inform.

If the player is on an enterable supporter or in an enterable container, trying to go somewhere will cause the parser to print out the message "You'll have to get off [the supporter] first." This is realistic, but a bit annoying, since the player knew what she wanted to do. Here's an easy way to fix it, suggested by Michael Callaghan:

```
Instead of going when the player is on a supporter (called S):
    say "(First getting off the [printed name of S])";
    surreptitiously move the player to the holder of S;
    continue the action.

Instead of going when the player is in a container (called C):
    say "(First getting out of the [printed name of C])";
    surreptitiously move the player to the holder of C;
    continue the action.
```

Moving Things Around

You can expect that during the game, the player will pick things up, carry them around, and drop them. But sometimes you need to move them yourself, in your own code. For instance, if the player rubs the magic lamp, you would probably want to move the genie into the room. The keyword for doing this is “now”:

```
Instead of rubbing the lamp:
	if the genie is off-stage:
		now the genie is in the location;
		say "Shazam! A genie appears!";
	otherwise:
		say "You make a small squeaking noise by rubbing the lamp."
```

In this case, “the location” refers to the room where the player is. If you need to move an object to a container or supporter, it’s usually easiest to refer to the container or supporter by name:

```
now the knockwurst is on the plate;
```

But sometimes you may not know exactly where the object should show up. That can happen, for instance, if you’re transforming an old shoe into a jewelled crown. In this case the shoe could be almost anywhere, so you need to figure out where it is, store that data, and then use the data to move the jewelled crown onstage:

```
let H be the holder of the old shoe;
move the jewelled crown to H;
remove the old shoe from play;
say "The old shoe turns into a jewelled crown!"
```

As the code above shows, you can’t move something off-stage by saying “now the X is off-stage”. The way to do it is to say “remove the X from play”.

For an example of how to move a bunch of indistinguishable objects at once using a loop, see p. 228.

I recently had a beginning student try to add an item to the player’s inventory (see below) by saying exactly that:

```
add the sword to the player's inventory. [Error!]
```

That won’t work, first because there is no container in the model world called “inventory” and second because “add” refers to an arithmetic operation, not to moving an object around in the world. The way to give the sword to the player as an item being carried is this:

```
now the player carries the sword.
```

Inventory

When the player types INVENTORY, INV, or simply I, Inform will print out a list of what the player is carrying or wearing. This list can be formatted in various ways — as a column, as a sentence, and so on. For details on how to do this, see **p. 6.7** of the *Recipe Book,* especially Example 176, "Equipment List."

If you're writing a game that tries to be realistic, allowing the player to carry 20 or 30 things at once is a bit silly (unless the player character is an alien with 20 or 30 hands). Some game authors prefer to limit the number of items the player can carry at once. As explained on **p. 3.19** of the Documentation ("Carrying capacity"), we can easily set this up by saying:

```
The carrying capacity of the player is 5.
```

But players don't like having a limited carrying capacity, because it's a huge hassle to have the game keep telling them, "You're carrying too many things already" when they try to pick up something new. The solution is to put a **holdall** container somewhere in the game — preferably in a place where the player will find it early in the game. The holdall could be a shopping bag, a backpack, a burlap sack, or whatever fits with your story. Creating a holdall is easy:

```
The backpack is an open container in the Barn. The backpack is a player's holdall.
```

If the player is carrying the holdall, when the player tries to pick up too many things at once, Inform will shuffle the excess items into the holdall automatically, like this:

```
>take apple
(putting the peach into the backpack to make room)
Taken.
```

Considerate Holdall is an Extension by Jon Ingold that improves on Inform's basic concept of a holdall. One thing it adds is the idea that certain things (such as a lighted torch) ought not to be shuffled into the holdall at any time.

If you want to make your game more realistic, you may want to think not just about the sheer number of items the player may want to pick up, but about their bulk. The Extension called Bulk Limiter by Eric Eve allows you to assign a bulk to any object and a bulk capacity to any container. This Extension is handy for a couple of reasons. With a little care, you can prevent silly things like having the player put a chair in his pocket. And if there are numerous bulky objects around (say, a chair, a string bass, and a shipping trunk), you can easily set the game up so that the player will only be able to carry one of them at a time. Your code might look something like this:

```
The carrying capacity of the player is 65.
The bulk of the chair is 45.
The bulk of the string bass is 50.
The bulk of the shipping trunk is 40.
```

Bulk Limiter is not a perfect solution to all problems of this sort, however. It doesn't give us any tools with which to handle long, thin objects that might not fit into a container. Also, if the player is prevented from picking up something bulky while carrying several things that are small, Bulk Limiter will just refuse the action; it won't shuffle the small things into a holdall automatically.

Things to Think About

In the simplest interactive fiction games, every portable thing in the game is useful for solving one puzzle. After the player has figured out that he can use the bent hairpin to unlock the jewel box, he can safely discard the hairpin, because it won't be needed again. Your game will be more interesting for players if you add variety to this scheme.

- Two or three of the things you create might have multiple uses.
- Two or three of your puzzles might have two solutions using different things.
- One or two things might be *red herrings.* They might not be good for anything at all.

Testing Where a Thing Is

In writing a game, it's often very useful to be able to test where something is. If the time bomb is in the suitcase, for instance, and the player is carrying the suitcase or just in the room with the suitcase, we might want to print out "You hear a faint ticking noise" once every few turns.

Inform has several words for describing and testing where things are. It's important to use the right word, because if you use the wrong one, your test may fail, causing a bug in your game. These words are ways of describing **relations**. Relations (see **p. 13.3** of the Documentation, "What are relations?", or p. 253 of the *Handbook*) are an important and versatile concept in advanced Inform programming.

Internally, your game has a *containment hierarchy*. This is a fancy way of saying that Inform knows when object A is inside of or on object B, while object B is inside of or on object C, while object C is … and so on. A room is always at the top of the hierarchy: it's not possible for one room to be inside another room — though we can fake this easily by creating a new room that's inside from another room. In this case, Inform understands that "inside" is just another direction, like north or down. This fact is mentioned on **p. 3.2** of the Documentation, "Rooms and the map."

If the aspirin tablet is in the pill box, the pill box is in the suitcase, the leather suitcase on the table, and the old oak table in the Dining Room, the containment hierarchy would look like this:

```
Dining Room
        old oak table
                leather suitcase
                        pill box
                                aspirin tablet
```

The words “in” and “on” mean just what you think they should — and they refer only to things that are *directly* in a container (or room) or on a supporter. In the hierarchy shown above, the table is in the Dining Room, but the leather suitcase is *not* in the Dining Room. Likewise, the suitcase is on the table, but the pill box is *not* on the table. Because Inform distinguishes “in” from “on,” the table is not “on” the Dining Room, and the suitcase is not “in” the table.

We can test whether the player (or another character) carries an item. Like on and in, “carries” only refers to things that the player carries directly. If the player carries the pill box, and the aspirin tablet is in the pill box, the result of the test “if the player carries the aspirin tablet” will be false.

Inform’s most general term for testing where a thing is is “enclosed by”. In the example above, the aspirin tablet is enclosed by *everything* above it in the containment hierarchy — the pill box, the leather suitcase, the old oak table, and the Dining room.

We can reverse these terms if we like. We can say, “if the Dining Room encloses the pill box”. This will be true if “if the pill box is enclosed by the Dining Room” is true.

The **location** of a thing is always the room, as **p. 3.25** of the Documentation (“The location of something”) points out. In the diagram above, the location of every object (except the Dining Room itself) is the Dining Room.

We can test whether two objects are sitting next to one another — in the same container, on the same supporter, carried by the same person, or in the same room — using the general-purpose term “holder”:

```
if the holder of the pear is the holder of the banana:
```

This condition will be true if they’re both carried by the player, or both in the same basket, or both on the floor of the room. But if the player is holding the basket and the pear, while the banana is in the basket, it will be false.

Things Can Have Properties

Often, an object can stay the same from the beginning of the game to the end. If the player finds a rock, for instance, that can be used to hammer a nail, probably not too much will happen to the rock during the course of the game. But it sometimes happens that we want to create an object whose state can change during the course of the game because of the player’s action. To keep track of what state the object is in, we need to give it a *property.* “Property” is simply computer programming jargon for an attribute or characteristic. To put it another way, the properties of an object are variables that are stored within the object — variables that may change depending on what the player does with or to the object.

Properties can be of two types. Some of them are numbers, while others are sets of words. They’re similar in some ways, but let’s look at them one at a time.

Number Properties

Attaching a number to an object (making it a property of the object) is very simple. To show how it works, we'll create a lamp that will run out of fuel after a certain number of turns:

```
The player carries a lamp. The lamp is lit. The lamp has a number called fuel-remaining. The fuel-remaining of the lamp is 50.

Every turn:
    if the fuel-remaining of the lamp is greater than 0:
        decrease the fuel-remaining of the lamp by 1;
        if the fuel-remaining of the lamp is 0:
            now the lamp is not lit;
            if the lamp is enclosed by the location:
                say "The lamp flickers and then goes out."
```

Number properties always have names — in this case, "fuel-remaining." We can manipulate them however we like. If this is an oil lamp, for instance, the player might be able to refill it from an oil can. Here's a somewhat oversimplified way to do exactly that:

```
The oil can is here. The oil can can be full or empty. The oil can is full.

Refilling is an action applying to one thing. Understand "fill [something]" and "refill [something]" as refilling.

Check refilling:
    say "[The noun] can't be refilled." instead.

Instead of refilling the lamp:
    if the oil can is enclosed by the location:
        if the oil can is full:
            now the oil can is empty;
            increase the fuel-remaining of the lamp by 50;
            say "You drain the oil can into the lamp[run paragraph on]";
            if the lamp is not lit:
                now the lamp is lit;
                say ", and quite magically the lamp's mantle begins glowing brightly
again[run paragraph on]";
            say ".";
        otherwise:
            say "The oil can appears to be empty.";
    otherwise:
        say "You have nothing to fill the lamp with."
```

This code is oversimplified because I've ignored the fact that the player would need a lighted match in order to light an oil lamp after it had burned out.

Word Properties

The easiest way to add word properties to an object is by simply listing the words. In this case, what we have is an anonymous (nameless) property. For instance:

```
The player carries a potato. The potato can be cold, warm, or hot.
```

Presumably the player will be able to do something during the game that will change the temperature of the potato. It’s good practice always to tell Inform what condition you want the object to be in at the start of the game. This removes any possible ambiguity, and makes your code easier to read:

```
The player carries a potato. The potato can be cold, warm, or hot. The potato is cold.
```

If you don’t do this, Inform will make an assumption — but its assumption may not be what you intended. Just to make our lives more interesting, if you give the potato’s anonymous property only two possible values (perhaps cold and hot) and fail to tell Inform what state the potato starts out in, the compiler will put the potato in the *second* of the two states. But if you give a list of three possible states, Inform will initialize the potato object in the *first* of the possible states.

The method you use to change the state of an object will, of course, depend on the nature of the object and also on the type of puzzle you’re implementing. Here’s a simple example:

```
The blazing fireplace is an open scenery container in the Library.

After inserting the potato into the blazing fireplace:
	now the potato is hot;
	say "You drop the potato into the blazing fireplace, and in a few moments the potato is glowing cherry red and smoldering cheerfully.";
	rule succeeds.

Instead of taking the potato when the potato is hot:
	say "You'd burn your fingers."
```

But letting the player do something that will make the potato hot is only the first step in implementing the puzzle. If the player should happen to refer to the object as a hot potato, the parser will report, “You can’t see any such thing.” So we need to instruct the parser about the vocabulary. Let’s omit, for now, the option of a warm potato, and create one that can be either cold or hot:

```
The potato can be cold or hot. The potato is cold. Understand "hot" as the potato when the potato is hot. The description of the potato is "It's brown and a bit lumpy[if hot]. It's also glowing with heat[end if]."
```

If there are several things in the game that might end up being too hot to pick up because they’ve been put in the fireplace, we might want to write a more general rule, along these lines:

```
Instead of taking something when the noun is hot:
	say "You'd burn your fingers." [Error!]
```

Unfortunately, this doesn’t work as expected. For some reason, Inform will assume that every object has the second value of the anonymous two-state property. Because we said “The potato can be cold or hot,” the parser will assume that *everything* is hot. We could get around this by saying, “The potato can be hot or cold,” but here’s a safer way to do it:

```
A thing can be hot or cold. A thing is usually cold.
```

Now all of the objects in your game will be explicitly cold, so the general-purpose Instead rule above will work with the potato, the poker, the gold doubloon, or anything else that you let the player put in the blazing fireplace.

But let’s suppose that you want some things to have a variable temperature, while other things don’t need this property. So you decide to be clever. You create temperature as a kind of value, like this:

```
Temperature is a kind of value. The temperatures are cold, warm, and hot.

The Library is a room. A toad is in the Library. The blazing fireplace is an open scenery container in the Library.

The player carries a potato. The potato has a temperature. Understand "hot" as the potato when the potato is hot. The description of the potato is "It's brown and a bit lumpy[if hot]. It's also glowing with heat[end if]."

After inserting the potato into the blazing fireplace:
	now the potato is hot;
	say "You drop the potato into the blazing fireplace, and in a few moments the potato is glowing cherry red and smoldering cheerfully.";
	rule succeeds.

Instead of taking something when the noun is hot:
	say "You'd burn your fingers." [Error!]
```

If you try this miniature game, it will compile, but when you test it with the command TAKE TOAD, you’ll get a run-time error:

```
>take toad

[** Programming error: tried to read (something) **]
Taken.
```

What has happened, in this case, is that the toad doesn’t have a temperature property, so trying to test whether it’s hot (using the Instead rule) can’t possibly work. You might think to try dodging the problem like this:

```
Instead of taking something:
	if the noun provides the temperature property:
		if the noun is hot:
			say "You'd burn your fingers.";
		otherwise:
```

```
            continue the action;
    otherwise:
        continue the action.
```

This looks perfectly sensible — but it won't compile. Why? If you look back at the code given a little earlier, you'll see that temperature isn't a property. It's a kind of value. Fortunately, there's a solution, which was suggested by Victor Gijsbers in a discussion in the newsgroup. We need to give the potato's temperature a distinct name, so that it isn't an anonymous property. Once the property has a name, we can test whether any object possesses that property. The final version is a bit more wordy, but it works as needed:

```
Temperature is a kind of value. The temperatures are cold, warm, and hot.

The Library is a room. A toad is in the Library. The blazing fireplace is an open scenery container in the Library.

The player carries a potato. The potato has a temperature called the heat. Understand "hot" as the potato when the heat of the potato is hot. The description of the potato is "It's brown and a bit lumpy[if the heat of the potato is hot]. It's also glowing with heat[end if]."

Instead of taking something:
    if the noun provides the property heat:
        if the heat of the noun is hot:
            say "You'd burn your fingers.";
        otherwise:
            continue the action;
    otherwise:
        continue the action.
```

Because we've named the property "heat", we can test whether any object that the player might refer to during the game possesses the property (called) heat.

Giving properties to objects is an extremely useful way of controlling how the objects will function in a game. Number properties are also useful; for more on this topic, you can consult **page 4.11**, "Values that vary," in the Documentation. In this section of the *Handbook* we've concentrated on properties that are lists of adjectives. We've seen how to create such a list within a single object, how to apply it to all objects, and how to create it as a separate kind of value that may or may not be applied to any given object. We've also looked at how to change the description of an object based on the current value of a property and how to let the parser know that a word can be used to describe the object only when the object's property has a certain value. Finally, we've shown how to give a property a name of its own, so as to be able to test whether the object has that sort of property.

Plurals & Collective Objects

As mentioned earlier (in Chapter 2), some objects need to be plural — for example, the tall old trees standing beside the forest path, or the cows grazing in a nearby field. It's usually not necessary to make every tree a separate object. Instead, we make a single scenery object, and give it the property plural-named. We can do this ourselves, by writing “The tall old trees are plural-named”, or we can let Inform figure it out, by using the word “some”. Let’s create a plural-named object that isn’t scenery:

```
Some scissors are on the sewing table. The description is "The scissors look quite sharp." Understand "sharp", "blades", and "shears" as the scissors.
```

Because we said “Some scissors”, Inform will make the scissors plural-named. So if it needs to mention the scissors — in an inventory list, for instance — it will say “You are carrying some scissors” rather than “You are carrying a scissors.”

If we need to create a collective object, such as sand or water, we can’t use “some” in this way. Here’s the wrong way to do it:

```
Some brackish water is in the pond. [Error!] The water is fixed in place.
```

If the player tries TAKE WATER, the game will respond, “Those are hardly portable.” Instead, we need to do it like this:

```
The brackish water is in the pond. The indefinite article of the brackish water is "some". The water is fixed in place.
```

When we do it this way, Inform understands that the water is not plural-named, but that it should nevertheless say “some water”, not “a water”.

Quite often, we need to write text that will produce outputs for various objects. In this case, we use “[the noun]” or “[a noun]” and let Inform substitute whatever noun is currently being talked about. (The substitutions “[the second noun]” and “[a second noun]” work the same way.) But when we do this, we have to be careful about how we construct the sentence. This looks correct, but it’s a bug:

```
say "[The noun] is too heavy for you to carry." [Error!]
```

If the noun being referred to at the moment happens to be plural-named, the output will be wrong:

```
>take boulders
The boulders is too heavy for you to carry.
```

We can fix this by hand, as shown in Chapter 2, like this:

```
say "[The noun] [if the noun is plural-named]are[otherwise]is[end if] too heavy for you to carry."
```

Inserting an if-test within text in this way is sometimes an ideal solution to a tricky problem. But when writing messages that may need to refer to things that are plural-named it’s usually more convenient to include the very handy Extension called Plurality by Emily Short. After including Plurality at the top of your Source, you can do this:

```
say "[The noun] [is-are] too heavy for you to carry."
```

Plurality has some other text substitutions that you’ll probably end up using often. See its

Documentation page to learn more.

Page 4.4 of the Documentation, “Duplicates,” shows how to create collections of indistinguishable items. This is occasionally useful. For example, you might be implementing an old-fashioned money system in which the player can have a purse containing copper pennies, silver dollars, and gold eagles. The coins within each group are identical, so it really doesn’t matter which object the player picks up when using the command PICK UP PENNY.

But precisely because they’re identical, writing code that will move one of them somewhere is slightly tricky. The way to do this is to refer to “a random” object of that kind, in the location where you’re sure one is to be found. Even if there’s only one object of the kind available, you still have to refer to “a random” object of that kind. (This syntax may change in a future version of Inform.)

Here’s a simple example:

```
A coin is a kind of thing.
A copper penny is a kind of coin.
A silver dollar is a kind of coin.
A gold eagle is a kind of coin.

The player carries three copper pennies, five silver dollars, and two gold eagles.

The shopkeeper is a man in the Poultry Shop. The shopkeeper carries a duck.

Instead of buying the duck:
        let P be a random copper penny carried by the player;
        now the shopkeeper carries P;
        now the player carries the duck;
        say "You buy the duck from the shopkeeper for a penny."
```

One thing that’s worth noting about this code is that you have to refer to objects of a kind using the *entire* term you used when creating the kind. If you write “let P be a random penny” after creating a kind called “copper penny”, Inform won’t know what you’re talking about.

Once you’ve created some indistinguishable objects, you’ll quickly discover that the game’s output looks a bit clumsy:

```
>drop dollars
silver dollar: Dropped.
silver dollar: Dropped.
silver dollar: Dropped.
silver dollar: Dropped.
silver dollar: Dropped.
```

There’s nothing really wrong with this except that it looks like a leftover from the 1980s. It doesn’t read well. Fortunately, there’s an easy fix: Download and install the extension Consolidated Multiple Actions by John Clemens. Including this extension in your game has no effect on the internal logic of the game (although it does require that the game be compiled to Glulx, so it can’t be used in .z5 and .z8 games), but it changes the way the action is reported to the player:

```
>drop dollars
You put down the five silver dollars.
```

More about Collections & Kinds

Inform lets you make either unique objects or kinds of objects. One reason to make several objects of a given kind is because they're indistinguishable from one another, like the pennies in the section "Plurals & Collective Objects." But sometimes we want to make several objects of a single kind that are similar, yet different — for example, the various kinds of fruit in the section "Things vs. Kinds," which starts on p. 78. When we do this, the player may very reasonably want to perform an action on all of the members of the kind at once (or at least, on all of the members that are available at that point in the game). Persuading Inform to report the action in a graceful way is not guaranteed to be simple.

The extension Consolidated Multiple Actions, as mentioned earlier, can handle some of these situations, but not all of them. If you want this extension to be used with any new actions you define in your game, however, you'll need to write a bit of extra code. This rather gross example shows how to do it:

```
Include Consolidated Multiple Actions by John Clemens.

The Test Lab is a room.

Moe is a man in the Lab.

A glop is a kind of thing. The indefinite article of a glop is "some". Understand "glop" as a glop. Understand "glop" as the plural of glop.

The jelly is a glop. The glue is a glop. The taco sauce is a glop. The player carries the jelly, the glue, and the taco sauce.

Smearing it on is an action applying to two things and requiring light. Understand "smear [things] on [something]" as smearing it on.

Report smearing it on:
	say "You smear [the noun] on [the second noun]."

Last for reporting consolidated actions rule when smearing on:
	say "You smear [consolidated objects] on [the second noun]."
```

The output looks like this:

```
>smear glop on moe
You smear the jelly, the glue and the taco sauce on Moe.
```

Consolidated Multiple Actions produces the output line — but in order for it to do its work, you have to do two things. First, your new action (in this example, the new action is called smearing it on) has to be defined using the “[things]” token, so that it can be used with multiple objects. Second, the action needs a “reporting consolidated actions rule”. Oddly enough, this rule has to be written for the action “smearing on” rather than the action “smearing it on” — don’t ask me why.

The third aspect of this example, creating a kind called glop, is only a nice extra. Even if we hadn’t done this, Consolidated Multiple Actions could handle a list of objects, like this:

```
>smear jelly and glue on moe
You smear the jelly and the glue on Moe.
```

There are also times when it’s convenient to create a fake “plural object” that can respond to certain player commands. Here’s an example:

```
The Guardhouse is a room.

A guard is a kind of person. Understand "guard" and "man" as a guard. The description of a guard is usually "He's wearing armor and a macrame nametag that says '[the noun].'"

Moe is a guard in the Guardhouse. Larry is a guard in the Guardhouse. Curly is a guard in the Guardhouse.

The plural-guards is a privately-named man in the Guardhouse. The plural-guards is scenery. Understand "guards" and "men" as the plural-guards. The printed name of the plural-guards is "guards".

Instead of examining the plural-guards:
        say "They're wearing armor and nametags, which identify them as Moe, Larry, and Curly."

Instead of doing anything other than examining with the plural-guards:
        say "You'll need to choose one or another of the guards and try again."
```

The vocabulary words “guards” and “men” will cause the parser to choose the plural-guards object. In this example, we’re using that object only to print out a collective description of the three guards; any other action will be intercepted by the “Instead of doing anything other than examining” rule. This type of effect can work well, but if the guards start moving around, it will get messy. For instance, there might be only one guard left in the guardhouse, in which case we would either have to remove the plural-guards object from play or write some code that would cause it to *pretend* that it didn’t exist — perhaps something along these lines:

```
Instead of examining the plural-guards:
        if the number of guards in the Guardhouse is 0:
                say "Guards? What guards?";
        otherwise: [and so on … more code would go here]
```

For a more complex example that shows how to consolidate the output messages when a collection of objects can get into different states, see Appendix B.

Objects That Have Parts

In real life, most objects are made up of various parts. For instance, an electric stove has heating elements (burners), perhaps an oven built into it, and knobs for turning the burners on and off. Inform lets us model a complex object like a stove by defining the other objects that are its parts.

With simple objects such as a knife or a cup, there's usually no need to create separate objects to model the parts. We can just make the names of the parts refer back to the main object, like this:

```
The player carries a knife. The description is "It's a shiny Bowie knife with a sharp three-inch blade and a black leather hilt." Understand "shiny", "Bowie", "sharp", "blade", "black", "leather", and "hilt" as the knife.
```

If the player types X BLADE or X HILT, the game will simply print out the description of the knife. That may be all that we need. But with a more complex object, adding parts is a good way of designing it. The basics of how to do this are well explained on **p. 3.23** of the Documentation ("Parts of things"). Example 34 ("Brown"), on that same page, shows how to make parts that can be detached and reattached. Unless our code detaches a part of an object, it will always be part of the object. If the larger object is picked up or dropped by the player, all of its parts will travel along with it automatically.

One of the advantages of using parts is that in Inform, any single object can be either a container or a supporter, but not both. If our model world includes an object like a dresser, we need to make the dresser itself a supporter (because the player may want to put things on top of it), and make its drawers openable containers. Making the drawers parts of the dresser is a wise precaution: If you should later change the design of the game to allow the dresser to be moved from place to place, the drawers will come along with it automatically.

Page 8.4 of the Recipe Book has some Examples showing how to make furniture. Example 80, "Yolk of Gold," has a complete implementation of a three-drawer dresser, with the added feature that the player will always find what he's looking for in the last drawer he opens, no matter which drawer it is.

There are two ways to make something part of something else. We can say:

```
The blade is part of the knife.
```

...or we can say:

```
The knife incorporates the blade.
```

Parts are detachable and attachable objects. This fact can be extremely handy. Let's suppose, for instance, that in your game you want the player to be able to put a stamp on a postcard:

```
The player carries a postcard and a stamp. Understand "card" and "post card" as the postcard. Understand "postage" as the stamp. The description of the postcard is "A plain white postcard[if the stamp is part of the postcard] with a stamp on it[end if]."

Instead of tying the stamp to the postcard:
	if the stamp is part of the postcard:
		say "You already did that.";
	otherwise:
		now the stamp is part of the postcard;
		say "You lick the stamp and affix it to the postcard."

Test me with "x card / fasten stamp to card / x card / take stamp".
```

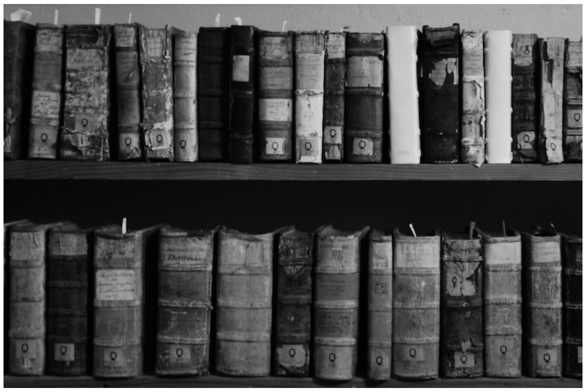
You may notice that the action provided by Inform is called tying it to. The player is unlikely to try TIE STAMP TO POSTCARD, but the command would work. The words ATTACH and FASTEN are used by Inform’s parser as synonyms for TIE. For suggestions on how to remove the stamp, consult Chapter 4, “Actions.”

The main point of the code above is that after the stamp has been attached to the postcard, the player will get this output:

```
>take stamp
That seems to be a part of the postcard.
```

In addition, when the player picks up the postcard and carries it around, the stamp will now be brought along for the ride. To learn how to detach parts of objects, see “Mr Potato Head” in Appendix B, p. 265.

Reading Matter



Inform’s standard rules assume that READ means the same thing as EXAMINE. This is not a bad assumption for a simple game. In the case of a roadside sign, the description of the sign would probably include the text printed on the sign, so there’s no need to have a special reading action. But in the case of a book or even a note on a piece of paper, we may want to make reading a separate action. Here’s how to do it:

```
A thing has some text called the reading-material. The reading-material of a thing is usually "".

The book is in the Library. The description is "A first edition of [italic type]In Praise of Folly[roman type], by Erasmus." Understand "praise", "in praise of", "folly", and "erasmus" as the book. The reading-material of the book is "A fascinating discussion of the idiocies to which the human mind is susceptible. After reading it, you feel quite humble, and even more foolish than before".

Understand the command "read" as something new.

Reading is an action applying to one thing and requiring light. Understand "read [something]" as reading.

Check reading:
	if the reading-material of the noun is "":
		say "Nothing is printed on [the noun]." instead.

Carry out reading:
	say "[reading-material of the noun]."
```

If your game also includes things like roadside signs, for which you want READ to give the same result as EXAMINE, you could change the Check rule like this:

```
Check reading:
	if the reading-material of the noun is "":
		try examining the noun instead.
```

Another way to do it would be to leave the Check rule saying "Nothing is printed on [the noun]" and add this for any signs:

```
Instead of reading the roadside sign:
	try examining the roadside sign.
```

My first idea, in designing this example, was to start by saying, "A thing can be legible or illegible. A thing is usually illegible." But after thinking about it for a minute, I realized I could simplify the code. All the Check Reading rule needs to do is find out whether the reading-material of the noun is "" (that is, whether it's empty). Anything that has text in its reading-material can now be read.

Another thing we might want to do in a game is create a book (or even a computer) in which the player can look things up. The best way to do this is by creating an action that uses the topics we want the player to look up — see "Actions with Topics" in Chapter 4 (p. 145).

Writing on Things

One of my students asked how to create a notepad that the player could write things on. A real software notepad, in which you could select and edit words and sentences, would be difficult to create in an Inform game, and quite likely impossible. But creating an object, such as a piece of paper or an old-fashioned slate, that the player can write on is not difficult.

An object like this might even be part of a puzzle: You could use it to let the player leave a message for another character. After writing this section of the *Handbook,* I expanded its ideas into an Extension called Notepad, which you can download from the Inform website. The Extension includes an example showing how to let a character respond to written commands. The example below is more basic than the Extension; it simply creates a slate that the player can write on or erase. This example uses a special type of text called indexed text (see p. 249). We'll first create a new kind of thing called a notepad. Then we'll change the command READ (which normally triggers the examining action) and add two new actions — writing it on and erasing.

```
Use maximum indexed text length of at least 2000.

A notepad is a kind of thing. A notepad has an indexed text called memo. The memo of a notepad is
```

```
usually "".

Understand the command "read" as something new.

Reading is an action applying to one thing and requiring light. Understand "read [something]" as reading.
Check reading:
    if the noun is not a notepad:
        say "Nothing is written on [the noun]." instead;
    otherwise if the memo of the noun is "":
        say "At the moment, [the noun] is blank." instead.
Carry out reading:
    if the memo of the noun is not "":
        say "On [the noun] you find the words '[memo of the noun].'";
    otherwise:
        say "Nothing is written on [the noun]."

Writing it on is an action applying to one topic and one thing and requiring light. Understand "write [text] on [something]" as writing it on.
Check writing it on:
    if the second noun is not a notepad:
        say "You can't write anything on [the second noun]!" instead;
    otherwise if the second noun is the slate:
        if the player does not carry the chalk:
            say "You'd need some chalk to do that." instead.
Carry out writing it on:
    let T be indexed text;
    let T be the topic understood;
    change the memo of the second noun to T;
    say "Writing '[T]' on [the second noun]."

Erasing is an action applying to one thing and requiring light. Understand "erase [something]" as erasing.
Check erasing:
    if the noun is not a notepad:
        say "There's nothing on [the noun] to erase." instead;
    otherwise if the memo of the noun is "":
        say "At the moment, nothing is written on [the noun]."
Carry out erasing:
    change the memo of the noun to "";
    say "You erase what was written on [the noun]."

The Lab is a room. A piece of chalk is in the Lab.

The player carries a fish. The description of the fish is "Scaly."
The player carries a slate. The slate is a notepad. The description of the slate is "Black."

Test me with "read fish / read slate / write E=mc2 on fish / write E=mc2 on slate / take chalk / write E=mc2 on slate / read slate / erase fish / erase slate / read slate".
```

Most of this example could be copied straight across into your own game. One detail that’s specific to the example is checking whether the player is carrying the chalk before letting her write on the slate. If your game uses a piece of paper as the notepad object, you'd probably want to change the piece of chalk to a pencil.

Mechanical Marvels

Inform provides a kind of thing called a device. The idea is, if you want to create something that can be switched on or off, you can call it a device. Inform will then understand that the commands SWITCH ON and SWITCH OFF can be applied to it (along with a few synonyms, such as TURN ON and TURN OFF). A device keeps track of whether it's switched on or switched off, so this property can be tested in your code:

```
if the electric razor is switched on:
        now the player is clean-shaven;
        [...and so on...]
```

The only other thing a device does, by default, is this: If the player examines it, the game will report whether it's currently switched on or switched off. That's okay if the device is something like an electric fan, which has a large black plastic on/off switch with the words ON and OFF printed on its mounting. It's less desirable if the device has no visible switch and doesn't look any different when it's switched on. In that case, we might prefer to prevent Inform's automatic mention of the device's on/off state. This can be done by removing the rule that causes the state to print out, like this:

```
The examine described devices rule is not listed in any rulebook.
```

If you do this, it's up to you to write a description for each device that will alert the player to the device's state:

```
The description of the electric razor is "It's a Remington cordless[if switched on]. At the moment it's
humming faintly[end if]."
```

Another odd thing about Inform's default implementation of devices — well, let's stick with the electric razor for a minute. The command SWITCH RAZOR means the same thing as SWITCH RAZOR ON. I personally prefer to have the command SWITCH RAZOR operate as a toggle: Giving the command when the razor is off should turn it on, and giving the command when the razor is on should turn it off. I managed to figure out one way to do this, and then Emily Short suggested a better way. In the interest of providing a more complete tutorial, let's look at them both.

My method requires a side trip to **p. 16.3** of the Documentation ("Overriding existing commands"), where you'll learn how to detach the word "switch" from the switching on action.

```
Understand the command "switch" as something new.
```

The tricky thing is, when we do this, SWITCH ON and SWITCH OFF won't work either, because we've detached the word "switch" from *all* commands. So in addition to creating a new action (which we'll call toggling), we also have to replace the grammar for SWITCH that we want to work the way it did before.

Here's the final version of my code:

```
Understand the command "switch" as something new.
Understand "switch [something] on" and "switch on [something]" as switching on.
Understand "switch [something] off" and "switch off [something]" as switching off.

Toggling is an action applying to one thing and requiring light. Understand "toggle [something]" and
"switch [something]" as toggling.

Check toggling:
	if the noun is not a device:
		say "[The noun] can't be toggled on and off."

Carry out toggling:
	if the noun is switched on:
		try switching off the noun instead;
	otherwise:
		try switching on the noun instead.
```

Now the command SWITCH RAZOR will alternately switch the razor on and off.

Emily's method is much simpler, and illustrates a cool feature of Inform programming:

```
Understand "switch [a switched on thing]" as switching off.
```

That's it — that's the whole answer. Emily explains it this way: "Because 'a switched on thing' is more specific than 'a thing', this understand line will be sorted early in the parse list and will be matched first if it applies [to the player's input]. Switched off things will continue to be caught by the existing understand line. In general, the clever use of adjectives in understand tokens is a very useful technique to have in one's Inform programming repertoire.... It's possible to tuck some complicated logic into the parser without having to write a separate action for each possible variation."

Creating buttons that can be pushed and levers that can be pulled is almost too simple to be worth mentioning:

```
The silver lever is a part of the shiny blinking plastic box.

Instead of pulling the silver lever:
	say "Clunk! You pull the lever, and a silver dollar drops into the hopper.";
	let SD be a random silver dollar in the money bin;
	now SD is in the hopper.
```

Example 283 in the Documentation, "Safety," shows how to make a spinning dial on a safe, but the Example is hardly complete. For one thing, the dial doesn't exist as a separate object that can be examined. And because it can't be examined, there's no way to find out what number it's currently set to. A recent question on the newsgroup rec.arts.int-fiction included some code for a dial that could be set to any number from 1 to 8. I modified the posted code slightly and came up with this dial:

```
The rotary dial is part of the safe. The dial has a number called current setting. The current setting of the dial is 1. The dial has a number called max setting. The max setting of the dial is 8. The description of the dial is "The rotary dial has eight numbers, 1 through [max setting]. At the moment it's set to [current setting]."

Instead of turning the dial:
    increase the current setting of the dial by 1;
    if the current setting of the dial > max setting of the dial:
        now the current setting of the dial is 1;
    say "You turn the dial to [current setting of the dial]."

Understand "set [something] to [number]" as setting the state of it to. Setting the state of it to is an action applying to one thing and one number. Understand "turn [something] to [number]" or "turn [something] to setting [number]" or "turn [something] to position [number]" or "adjust [something] to [number]" or "adjust [something] to position [number]" or "adjust [something] to setting [number]" as setting the state of it to.

Check setting the state of it to:
    if the noun is not the rotary dial:
        say "You can't set [the noun] to a number." instead;
    if the number understood < 1, say "The lowest setting is 1." instead;
    if the number understood > max setting of the dial, say "Sorry, the dial can only be set from 1 to [max setting of the dial]." instead.

Carry out setting the state of it to:
    now the current setting of the noun is the number understood;
    say "You turn the dial to [number understood]."
```

In a real game, you might want to create three dials of this type, and mount them side by side on the door of the safe. This would force the player to figure out what three-digit number to dial. You might also want to write some code that would interpret DIAL 123 as a command to set the first dial to 1, the second dial to 2, and so on. This would take a bit of work, so I'll leave it as an exercise for you to try on your own. The main things to notice about the code given above are:

1) A new action, setting the state of it to, can be used to SET DIAL TO 3 (or any other number between 1 and the max setting of the dial).
2) The command TURN DIAL will increment (add 1 to) the setting of the dial.
3) If the dial is already at its max setting, TURN DIAL will rotate it around to 1 again.

Many other kinds of mechanical contrivances might be useful in your game. If you want to create a tricycle that the player can ride, for instance, you'll want to look at the Extension called Rideable Vehicles by Graham Nelson. In Appendix B of this *Handbook,* you'll find a neat example of a device that will respond to commands.

Smelling & Listening

Inform includes the commands “smell [something]” and “listen to [something]”, but they don’t do anything. The output is “You smell nothing unexpected” or “You hear nothing unexpected.”

An important part of writing good fiction (of any kind, not just interactive fiction) is letting your readers use all of their senses. Adding an odor or sound to a single object with an Instead rule is easy:

```
Instead of smelling the garbage:
	say "It smells awful!"
```

This will work if the player types SMELL GARBAGE. But the player might just type SMELL. As explained on **p. 7. 7** of the Documentation, “The other four senses,” this command is redirected to smelling the location (the room) — and again, by default, the game replies, “You smell nothing unexpected.” If the garbage is in the location, this response is just plain wrong. If the garbage is fixed in place, we can give the room an Instead rule so that it will respond to the SMELL command:

```
Instead of smelling the Alley:
	say "It smells of garbage."
```

But we’ll have to do that for every room where there’s an odor, and if some of the things that have odors are getting moved around during the game, keeping track of them in order to list the odors will get messy.

Fortunately, it’s not hard to improve Inform’s handling of smelling and listening. The *Recipe Book* provides several good examples of ways to add sounds to a game, so in the *Handbook* we’ll concentrate on odors. In the code below, we’re going to give every thing in the game a scent (which is a text in quotes). By default, this text will be empty. But now we can add a scent as a property of a thing, rather than needing to write a whole new Instead rule. A single Instead rule (Instead of smelling something) will handle any object the player tries to SMELL.

The other feature we’ll add is this: We’ll define a thing as smelly if its scent is not “” (that is, when the scent property is not empty). When there is something smelly in the room, typing SMELL will list the smelly things in the room.

```
A thing has some text called the scent.

Definition: a thing is smelly if its scent is not "".

Instead of smelling something:
	if the noun is smelly:
		say "[scent of the noun][paragraph break]";
	otherwise:
		say "It smells like an ordinary [noun]."

Instead of smelling a room:
	let L be the list of smelly things enclosed by the location;
```

```
    if L is empty:
        say "You smell nothing unexpected.";
    otherwise:
        say "You smell [a list of smelly things enclosed by the location]."

The Kitchen is a room.
The banana is in the Kitchen. The scent of the banana is "It smells sweet and ripe."
The loaf of bread is in the Kitchen. The scent of the bread is "Ah -- fresh-baked bread."
```

Here's the output when the player is in the Kitchen:

```
>smell
You smell a banana and a loaf of bread.
```

There are other ways to implement smelling. This is normal in writing interactive fiction — the author can usually solve the same problem by using several different techniques. Below is a simple example game that I've adapted from an example uploaded to the ifwiki (www.ifwiki.org). It differs from the code shown above in a number of ways.

First, it implements the smelling action in a more complete way. Because it uses Carry Out and Report rules rather than Instead rules, it has to start by removing the block smelling rule from the Standard Rules. Second, it gives distinct odors to rooms rather than just listing any smelly objects that are in the room. Third, it has the beginnings of a puzzle, in the form of a spacesuit that will block smelling. Fourth, the odor of the sewage is so powerful that it will prevent you from smelling the flower.

```
[First we'll create the property for all things and rooms.]
A thing has a text called odor.
A room has a text called odor.

[We need to remove the default blocking rule.]
The block smelling rule is not listed in the check smelling rulebook.

[Now we'll add a default carry out rule.]
The last carry out smelling rule:
    say "[if the odor of the noun is not empty][the odor of the noun][paragraph break]
[otherwise]You smell nothing unexpected.[end if]".

[For completeness we'll add a rule for NPCs smelling things.]
Report someone smelling:
    say "[The person asked] smells [the noun]."

[The player might try 'smell the odor', so we'll allow that by creating a backdrop:]
The ambient-odor is a backdrop. It is everywhere. The description is "The odor is impalpable."
Understand "odor", "odour", "stench", "stink", "fragrance", "reek", and "smell" as the ambient-odor.
Instead of doing anything other than smelling or examining with the ambient-odor:
    try examining the ambient-odor.
Instead of smelling the ambient-odor:
    try smelling the location.

Street is a room. "You are in a street. The sewer is below you." The odor is "You pick up a faint odor
```

```
from below."
A flower is here. The description is "It's just a nice flower. You don't know what type." The odor is "It smells wonderful."
A spacesuit is here. It is wearable. Understand "space suit" and "suit" as the spacesuit. The description is "Spacesuits are wonderful things, but they make EVERYONE look fat."

Sewer is below Street. "You are in a sewer. The street is above you."
Some sewage is here. It is fixed in place. The description is "Horrible smelly sewage is everywhere in the sewer." The odor is "It reeks."

[Finally we'll add some rules to restrict smelling.]
Instead of smelling when the spacesuit is worn and the noun is not yourself and the noun is not the spacesuit:
	say "You can't smell anything outside the spacesuit while wearing the spacesuit."
Instead of smelling in the presence of the sewage when the spacesuit is not worn and the noun is not the sewage:
	say "The disgusting reek of the sewage overwhelms your nose. You can't smell anything else."
```

In passing, you might want to take note of the lists of conditions in those last two Instead rules. This syntax is not used much in the *Handbook,* but it's good to know about it.

Transforming One Thing into Another

In a game that includes magic, you might want to turn an object into another object. (Maybe this would happen when you cast a spell on the object using my Spellcasting Extension.) The easy way to do this in Inform is to create both objects but leave one of them offstage at the beginning of the game. When it's time to transform the object, simply whisk it offstage and replace it with the other object. Here's the classic story situation, more or less:

```
The Creekside is a room. "A burbling creek runs through the forest here."

The toad is an animal in the Creekside. "An ugly, warty toad squats on a rock near the creek." The description is "He's ugly and green." Understand "ugly" and "green" as the toad.

The princess is a woman. "A beautiful princess stands before you!" The description is "She's the most beautiful woman you've ever seen." Understand "beautiful" and "woman" as the princess.

Instead of kissing the toad:
	now the princess is in the location of the toad;
	remove the toad from play;
	say "As your lips touch the ugly toad, it shimmers and sparkles and turns into a beautiful princess. 'My goodness,' the princess declares. 'Thank you!'"
```

Initially the princess is nowhere. The Instead rule does three things. First, it makes a note about where the toad is. This is a good idea, because in a real game the toad might be moving around, and it would be a bug if you kiss the toad in the throne room and have the princess pop up back at the creekside. Then the rule gets rid of the toad and moves the princess to the room where the toad was.

If we're transforming one inanimate object into another, we'll want to do it just a bit differently, because an inanimate object might be on a supporter or in a container. If you're transforming a gold crown into an old shoe, for instance, and if the crown is on a table, you don't want the shoe to appear on the floor of the room — you want it to show up on the table. To take care of this, we need to refer to "the holder" of the crown. The holder will always be the crown's immediate location:

```
Instead of casting shazam at the crown:
	now the old shoe is in the holder of the crown;
	remove the crown from play;
	say "The crown emits an unhappy crinkling, shriveling sort of noise, and turns into an old shoe."
```

New Ways to Use Things

Inform's model of how things exist (and can be used) in the world is very simple. Sometimes you'll want the player to be able to do something new or unexpected with a thing. In this section we'll borrow some ideas from Chapter 4, "Actions," and show you how to allow new kinds of actions with the objects in your game. For details on how Check, Carry Out, Report, and Instead rules work, consult Chapter 4. The main purpose of the section you're now reading is to show how to do interesting things with the new objects you've created.

This section was inspired by an old post on rec.arts.int-fiction, which I spotted while poking around in the ifwiki (www.ifwiki.org). The list of new actions suggested by Jan Thorsby in that post was short but suggestive: holding an object close to or against another object, holding an object up in the air, threatening an NPC with an object, or tipping a bookshelf forward so that the thing on the top shelf will fall down. I'll leave you to explore the process of threatening an NPC for yourself; it's not difficult. The other three actions are illustrated below.

The number of new things you might want players to be able to do is probably infinite. Some of them will turn out to be easy to create; others may be surprisingly tricky for you to implement. The key, in each case, is to think logically and try to handle all of the things a player might do. You'll also need to consider all of the states that your objects might get themselves into. For example, you may know that you only want your new tipping action to apply to one object in the game — but once players know that the TIP command works, they'll surely try it on anything and everything. So you need to write a check rule for your new action that will handle any and all of players' attempts with a sensible-sounding refusal message.

Holding Something Up in the Air

The player might very reasonably want to hold an object up in the air so as to make it visible at a distance. You might want to hold up a white sheet, for instance, if you're marooned on an island and trying to attract the attention of a passing ship. Below is a simple example game that implements this action. Notice the long list of input grammar that is allowed. You'd probably want to add even more grammar lines to this list if you're using this action in a game.

The key point in this example is that the sheet has to be held aloft for three consecutive turns on the beach in order to signal the passing ship. If the player drops the sheet or walks west into the jungle, the sheet is no longer aloft, and the count drops back to zero. Taking care of details like these will add realism to your game.

The details are taken care of by the after dropping something and after going rules, and by the new property called aloft. Every object in the game can be aloft or not — but most of them won't be, so this new property will only be useful in a few cases.

Why make aloft a general property of all things, when you're only going to be using it for one object? Because it reduces the chance of bugs, and makes your code easier to maintain and expand. If you should later be writing a different section and decide that the player needs to hold a torch aloft, you don't have to go back and revise anything, because the torch already has that property.

```
A thing can be aloft. A thing is usually not aloft. A thing has a number called the aloft-count. The aloft-count of a thing is usually 0.

The Beach is a room. The player carries a white sheet.

Elevating is an action applying to one thing. Understand "hold [something preferably held] up", "hold up [something preferably held]", "wave [something preferably held]", "elevate [something preferably held]", "hold [something preferably held] aloft", "hold [something preferably held] in the air", "hold [something preferably held] in air", and "hold [something preferably held] up in the air" as elevating.

Check elevating:
        if the noun is not held:
                say "You'll need to pick up [the noun] first.";
                rule fails.

Carry out elevating:
        now the noun is aloft;
        say "You hold [the noun] high in the air."

After dropping something:
        now the noun is not aloft;
        now the aloft-count of the noun is 0;
        continue the action.

After going:
        repeat with item running through things carried by the player:
                now item is not aloft;
                now the aloft-count of item is 0;
        continue the action.

The Jungle is west of the Beach.

Every turn:
        if the white sheet is aloft and the location is the Beach:
                increase the aloft-count of the white sheet by 1;
        if aloft-count of the white sheet is 4:
                say "A passing ship spots your signal, and steers in toward shore. You're saved!";
```

```
		end the game in victory.
```

Tipping Something

Placing an object out of reach — on a high shelf or down at the bottom of a well, for instance — is a standard puzzle in IF (see Chapter XX, “Puzzles”). Some high shelves are fixed to the wall, but some might be the top shelf in a heavy bookcase that can’t be moved, but can be rocked or tipped. In the example below, note that the pushing action (which is part of Inform’s standard library) has been remapped to the new rocking action.

```
The Living Room is a room. "The only furniture here is a tall bookshelf."

The bookshelf is a scenery supporter in the Living Room. "It's so tall you can't possibly reach the top shelf -- and oddly enough, there's only one shelf, the top one." Understand "top", "shelf", and "bookcase" as the bookshelf.

The Ming vase is on the bookshelf. The description is "A priceless vase!"

Instead of putting anything on the bookshelf:
	say "You can't reach the top shelf."

Instead of taking something when the noun is on the bookshelf:
	say "You can't reach [the noun], as the shelf is too high."

Rocking is an action applying to one thing and requiring light. Understand "tip [something]", "rock [something]", "shake [something]", "tip [something] forward", and "rock [something] forward" as rocking.

Check rocking:
	say "There's no need to agitate [the noun]."

Instead of rocking the bookshelf:
	if the Ming vase is on the bookshelf:
		now the player carries the Ming vase;
		say "You give the shelf a good shake ... and the vase teeters forward, reaches the edge, and plummets! Fortunately, you're able to catch it just before it hits the floor.";
	otherwise:
		say "There's nothing else up there."

Instead of pushing the bookshelf:
	try rocking the bookshelf.
```

Holding Something Against Something Else

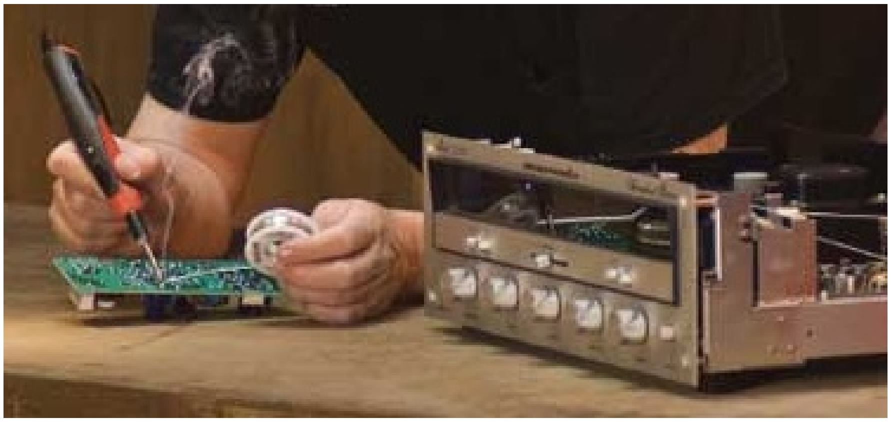
This example creates a new action, holding it against. This action has two Report rules. The thing you need to know is that when your game is being compiled, Inform assembles all of the rules into rulebooks. *Both* of these rules will end up in the report holding it against rulebook — but Inform uses some very specific instructions (let’s not call them rules, as that would be confusing) about how to do this. It puts more specific rules near the top of the rulebook, and more general rules later in the rulebook.

As a result, the Report holding the stethoscope against the wall rule will be earlier in the “report holding it against” rulebook. When this rulebook is being consulted (after the player has used the command HOLD STETHOSCOPE AGAINST WALL, for instance), Inform will consult the rulebook *until one of the rules makes a decision*. Then it will stop. For this reason, we’ll end the specific rule with the line “rule succeeds.” This will prevent the more general rule from being applied.

```
The Cell is a room. "A dank, rat-infested cell. Faint murmurings can be heard from the other side of the wall."

A stethoscope is here. The stethoscope is wearable. Understand "scope" as the stethoscope.

Report wearing the stethoscope:
	say "You insert the ear-pieces into your ears.";
	rule succeeds.

The wall is scenery in the Cell. The description is "Solid cement blocks."

Holding it against is an action applying to two things. Understand "hold [something] against [something]", "press [something] against [something]", and "apply [something] to [something]" as holding it against.

Check holding it against:
	if the player does not enclose the noun:
		say "You'll need to be holding [the noun] in order to do that.";
		rule fails.

Report holding it against:
	say "You press [the noun] against [the second noun] for a moment, but nothing seems to have been accomplished by this."

Report holding the stethoscope against the wall:
	if the player wears the stethoscope:
		say "You press the stethoscope against the wall. [one of]After a moment you hear a man's voice as distinctly as if he were in the cell with you: 'Good thing he doesn't know I hid the key in the light fixture!'[or]In the next room, two men are discussing horse racing. Since you already know where the key is hidden, there's probably no need to keep listening.[stopping]";
		rule succeeds.
```

Notice the non-default message that’s printed out when the player wears the stethoscope. Also notice the use of “[one of] … [or] … [stopping]” to insure that the conversation about the hidden key is printed out only once.

Chapter 4: Actions



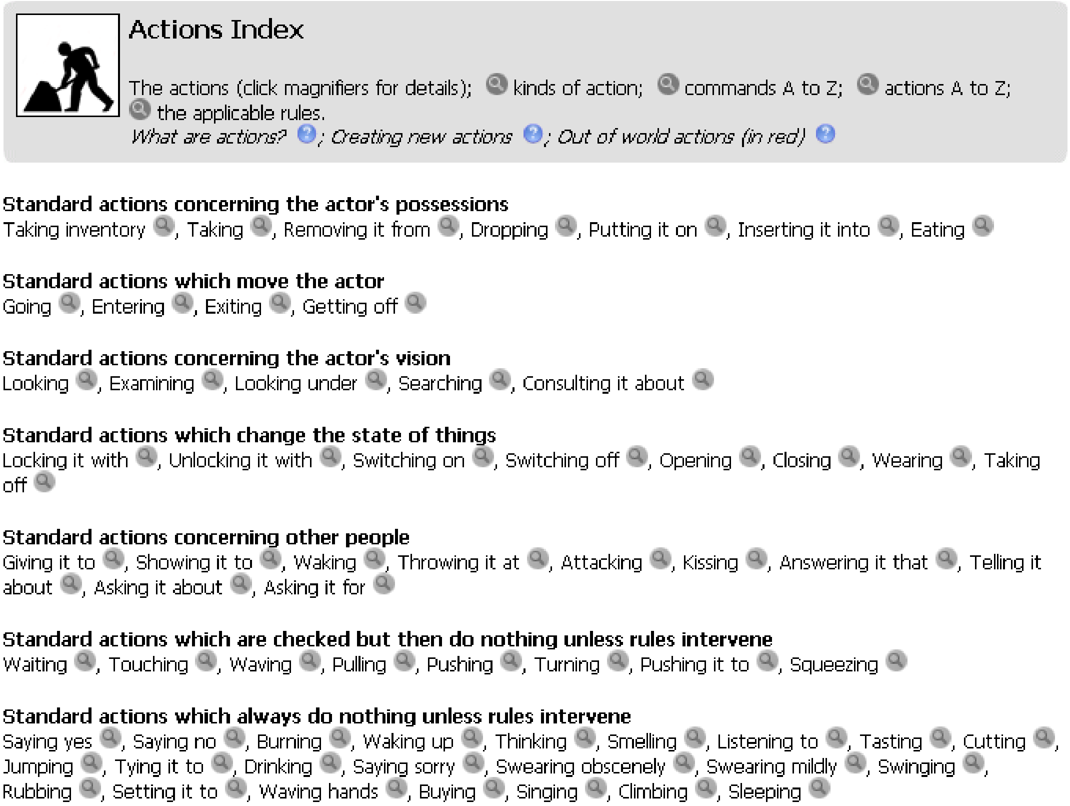
An action is what happens within the game when the player types a command. Inform provides a bunch of built-in actions, but before long you'll probably want to modify the built-in actions, or create some new ones. **Page 12.7** in the Inform Documentation "New actions," begins this way: "It is not often that we need to create new actions...." In my own experience writing games, this is simply not true. Inform's built-in actions handle many of the basic types of commands that the player may want to use, but many equally common actions are not included. Soon after I add a few objects to a new game, I begin to realize that the player may want to try all sorts of odd commands with the objects. Even if I only want to give the player an interesting "you can't do that" message, I still need to create a new action.

As a general rule, you should try to anticipate *all* of the command words that your players may think to try, and write code to handle them. Your beta-testers will be a great source of feedback on this point: If you read the game transcripts they send you, you'll probably spot a dozen or more synonyms for actions that you have implemented, plus an assortment of plausible actions that you may want to add. Some of these might even suggest alternate solutions to your puzzles. Adding a few alternate solutions is not just a courtesy to players: It also deepens and enriches the model world of your game.

Built-In Actions

From the moment you start writing your first interactive story, you'll be able to use a handy set of actions that are built into Inform. To see the list of built-in actions, open up your game (or create a new game and add as little as a single room) and click the Go button. After the game compiles, go to the Index tab and click on the Actions header. You'll find a list of about 50 actions (see next page).

With no extra effort on your part, the player will be able to move from place to place, examine objects, pick them up, take inventory, drop objects that are being carried, put objects on top of supporters or in containers, open and close doors, wear and remove clothing, unlock or lock things that are locked, and so on.



Actions Index

The actions (click magnifiers for details); kinds of action; commands A to Z; actions A to Z; the applicable rules.
What are actions?; Creating new actions; Out of world actions (in red)

Standard actions concerning the actor's possessions
Taking inventory, Taking, Removing it from, Dropping, Putting it on, Inserting it into, Eating

Standard actions which move the actor
Going, Entering, Exiting, Getting off

Standard actions concerning the actor's vision
Looking, Examining, Looking under, Searching, Consulting it about

Standard actions which change the state of things
Locking it with, Unlocking it with, Switching on, Switching off, Opening, Closing, Wearing, Taking off

Standard actions concerning other people
Giving it to, Showing it to, Waking, Throwing it at, Attacking, Kissing, Answering it that, Telling it about, Asking it about, Asking it for

Standard actions which are checked but then do nothing unless rules intervene
Waiting, Touching, Waving, Pulling, Pushing, Turning, Pushing it to, Squeezing

Standard actions which always do nothing unless rules intervene
Saying yes, Saying no, Burning, Waking up, Thinking, Smelling, Listening to, Tasting, Cutting, Jumping, Tying it to, Drinking, Saying sorry, Swearing obscenely, Swearing mildly, Swinging, Rubbing, Setting it to, Waving hands, Buying, Singing, Climbing, Sleeping

If you click on the gray magnifying-glass button next to an action in the Actions Index, a page will open up in which you'll learn more about that action. On the page describing an action, you'll see a list of typed commands that can cause the action to happen and a list of rules that can affect the outcome after the action starts. This page doesn't list exact synonyms (for instance, "discard" is not listed on the page for the drop action), but there are two ways to figure out what actions are being generated by a given command word.

First, the SHOWVERB debugging command can be used in the game to display a list of the Inform 6 grammar lines that cause the action to happen. (For more on the relationship between Inform 6 and Inform 7, see p. 257.) Here, for instance, is the output for the drop action:

```
>showverb drop
Verb 'discard' 'drop' 'throw'
     * multiheld -> Drop
     * held 'at' / 'against' / 'on' / 'onto' noun -> ThrowAt
     * multiexcept 'in' / 'into' / 'down' noun -> Insert
     * multiexcept 'on' / 'onto' noun -> PutOn
```

Don't worry too much about the picky details of that output. (Inform 6 programmers will understand it at a glance.) The main point is that 'discard', 'drop', and 'throw' are all synonyms for an action — and all three words can lead to any of four different actions (Drop, ThrowAt, Insert, or PutOn).

Second, if you scroll down on the Actions page, you'll find a list headed "Commands available to the player." If you're trying to find out what action(s) can be generated by a command word, such as

DISCARD, you can look it up in this list. You’ll find that it says, “same as drop.” And the four listings for DROP are exactly what’s shown above in the Inform 6 code: dropping, throwing it at, inserting it into, and putting it on.

Another useful feature of Inform is the ACTIONS debugging command. In debug mode only (that’s the mode you’re using as you work on your game), giving the ACTIONS command in the game turns on some special output that reveals more about what’s going on behind the scenes in the software. Here’s an example of how the ACTIONS debugging command works:

```
You can see an apple here.

>actions
Actions listing on.

>pick up apple
[taking the apple]
Taken.
[taking the apple - succeeded]
```

Here we find a listing for the action (the action of *taking* with the apple as the noun), and also an indication that the action’s rules succeeded. Next, we’ll look at a slightly more interesting example. Here’s the action listing in the game for an absurd action, one that isn’t going to work:

```
>eat the book
[eating the book]
(first taking the book)
[(1) taking the book - silently]
[(1) taking the book - silently - succeeded]

That's plainly inedible.
[eating the book - failed the can't eat unless edible rule]
```

The action listing reveals two things. First, the command EAT triggers the *implicit* action of taking. This is because Inform assumes that the player will need to be holding an object in order to eat it. (If your game involves a pie-eating contest in which the player has to eat pie while her hands are tied behind her back, you’ll need to work out a way to override this implicit action. It can be done using a Before rule, as explained later in this chapter.) Second, while the action of taking the book implicitly succeeded, the action of eating the book failed. Inform tells us not only that it failed, but what specific rule caused it to fail. This information could be important if we want to replace that rule. Replacing the rules in the Standard Library is an advanced topic, however. It’s touched on only briefly in this *Handbook,* in Chapter 10.

One of the common sources of frustration for newcomers to Inform is not knowing how to refer to the built-in actions in their code. One of the most frequent problems comes from the inserting it into action. Newcomers tend to try to refer to this action in what seems to be a natural way, like this:

```
Instead of putting the apple in the basket: say "The basket scoots away from you!" [Error!]
```

But that doesn't work, because there is no action called "putting it in." Here's the correct way to refer to this action:

```
Instead of inserting the apple into the basket: say "The basket scoots away from you!"
```

By using the ACTIONS command and then using a command such as PUT APPLE IN BASKET in your game, you can see which action or actions are being triggered.

Using "Instead"

The easiest way to start customizing how Inform handles actions is to write a few Instead rules. We're going to create a bubbling beaker of magic potion and then use a couple of Instead rules on it. Inform already has actions for drinking and taking. The command DRINK POTION, however, produces the output, "There's nothing suitable to drink here," which is not what we want in this particular case. TAKE POTION will allow the player to pick up the potion, and that might not be what we want either, for one reason or another. Here's how to customize Inform's default behavior using Instead rules:

```
The bubbling potion is on the stone table. The description is "The bubbling beaker is full of noxious-looking yellow potion." Understand "beaker", "yellow", and "noxious" as the bubbling potion.

Instead of drinking the bubbling potion:
	say "You take a tiny sip, but it smells awful, so you put it down again."

Instead of taking the bubbling potion:
	say "If you try to carry it around, you'll only spill it and stain your clothes."
```

The response above to TAKE POTION will prevent the player from picking up the potion, because the Instead rule will stop the taking action in its tracks. We could do the same thing by saying, "The bubbling potion is fixed in place", but that produces a boring message when the player tries TAKE POTION. The code above will give a more interesting message.

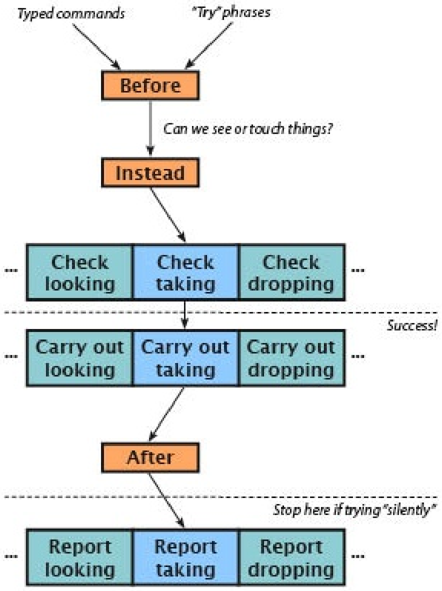
Likewise, if the player tries DRINK POTION, a new message will be printed out in place of the default message. Nothing will actually happen in the model world as a result of this command — but if we wanted something to happen, we could just add it to the Instead rule. In an extreme case, we might want to do something like this:

```
Instead of drinking the bubbling potion:
	say "Your first tiny sip of the bubbling potion causes your whole body to be wracked by horrible spasms.";
	end the game in death.
```

Almost anything we might want to happen in the model world can be tucked into an Instead rule like this. To understand how Instead rules work, we need to delve deeper into Inform's action processing.

Action Processing

Page 12.2 in the Documentation, “How actions are processed,” has a diagram (a flowchart) that shows — as you might expect — how actions are processed. The first time I looked at this diagram, it baffled me, but it really does make sense. (See page 146 for a detailed run-through of this diagram).



When the player types a command, the game goes through a number of steps. First it makes sure it knows what the input words are, and that the command makes grammatical sense. (This step isn’t in the diagram.) If there’s a typo in the input — or even if the parser knows all of the individual words but doesn’t understand how they’re strung together — the parser will reject the input before anything happens. Assuming the parser can figure out what the words are, and that they’re in a grammatical form that the parser knows, Inform starts processing the action.

The first step shown in the diagram is Inform looking to see if there is a Before rule that might affect the action. But in fact there’s a step before this, which is also not shown in the diagram. If the action includes a noun, as most actions do — PICK UP THE APPLE, for instance — the game first checks to make sure that there’s an object in the room that can be referred to as “apple.” If the apple is in a different room, it’s not in scope. Scope checking, which is explained briefly on p. 38 of this *Handbook,* is one of the most important things the parser does, and it happens *before* the Before rules are considered. (There’s an exception to this, as we’ll see in the section below on “[Any Thing]”.)

It’s easy to write your own Before rules for existing actions — but it’s also easy to create odd bugs with Before rules. While learning to use Inform, you should probably stick with Instead rules when you need to alter the processing of a built-in action. As the diagram in 12.2 shows, *after* the Before rules are considered, Inform pauses to figure out whether the objects being referred to are actually available for action processing — whether they can be touched. This is a narrower question than the question of whether an object is in scope. If an object is in a transparent container in the room, for instance, and if the container is closed and locked, the player can see what’s in the container, so objects in the container are in scope. But the objects in the container can’t actually be touched or taken. While the Before rules are being considered, Inform doesn't yet know that the glass container is closed and locked! Here's an example of a Before rule that may look sensible at first glance, but that will get you in trouble:

```
The glass case is an openable lockable transparent container in the Lab. The glass case is closed and locked.

An apple is in the glass case. A wicker basket is an open container in the Lab.

Before taking the apple:
    if the player does not carry the apple:
        if the player carries the basket:
            if the apple is not in the basket:
                now the apple is in the basket;
                say "You pick up the apple and put it in the basket.";
                rule succeeds.
```

The author of this code might be trying to save the player a little work, by putting the apple into the basket automatically in response to the command TAKE APPLE. But can you see what's going to happen? The Before rule tests several things, but it doesn't test to make sure the apple is actually available for taking. And Inform doesn't yet know that either, because the test for availability won't be run until *after* the Before rule is considered. The result is a bug — a player who is carrying the basket can get the apple out of the glass case without unlocking the case!

```
>take apple
The glass case isn't open.

>take basket
Taken.

>take apple
You pick up the apple and put it in the basket.

>i
You are carrying:
  a wicker basket
    an apple
```

This illustrates why Before rules can be a little treacherous. Instead rules are safer.

But sometimes a Before rule will give you better results. Consider this code:

```
The chocolate cake is on the table. It is edible.
Instead of eating the chocolate cake:
    say "But you're trying to diet!"
```

If you use an Instead rule here, Inform will perform an implicit taking action, as shown earlier in this chapter. The implicit taking action will cause the player to pick up the cake before refusing to allow the cake to be eaten. This happens because Inform assumes that things can only be eaten if they're being held.

```
>eat cake
(first taking the chocolate cake)
But you're trying to diet!

>i
You are carrying:
  a chocolate cake
```

A Before rule will run before the implicit take action happens, so the player won't automatically pick up the cake before changing her mind about eating it:

```
Before eating the chocolate cake:
	say "But you're trying to diet!" instead.
```

Note the use of the word "instead" at the end of the Before rule above. This may look odd, since we're writing a Before rule, not an Instead rule, but it's a convenient syntax. In this case, what "instead" does is cause the Before rule to succeed (see "Rulebooks & 'Stop the Action'," below), thus shutting off any further processing of the player's command. If you forget to add "instead", a Before rule *doesn't* shut off the action processing.

If that doesn't make sense yet, keep reading. It will start to make sense — and it's important.

For technical reasons, there's only one Before rulebook and one Instead rulebook. As an action is being processed, Inform will run through these rulebooks looking for rules that might apply. Any rule that applies might end with a "rule succeeds" line, which will shut off the processing. If none of the Before rules that Inform thinks might apply ends with a "rule succeeds" or "rule fails" (which in this case is pretty much the same thing), Inform proceeds to the Instead rulebook, and then to the Check rulebook for the specific action. Each action has its own Check, Carry Out, and Report rules. Before the Report rules for the action are considered, though, Inform dips into the After rulebook (again, there's only one, though it might contain dozens of rules applying to different actions). This is what the diagram on **p. 12.2** is illustrating. If no After rule shuts off the action processing, the Report rules for the current action will be considered.

This cycle — Check, Carry Out, Report — is the nuts and bolts of how Inform processes actions. They're what happens with built-in actions that are defined in the standard library. Before, Instead, and After are mostly for the author to use, though Inform has some built-in Before rules. You can also write your own Check, Carry Out, and Report rules, as we'll see later in this chapter.

In a Check rule, Inform checks whether the action makes sense. The eating action (built-in) gives a good example of the use of a Check rule. If you haven't told Inform that an object is edible, when the player tries to EAT it, the "can't eat unless edible rule," which is a Check rule for the eating action, will print out the message "That's plainly inedible." This rule will then fail (it ends with the line "rule fails"), which will stop Inform from going on to the Carry Out rulebook for the eating action.

If no Check rule gets in the way, the Carry Out rules will run next. The purpose of a Carry Out rule is to change the model world in whatever way is needed. In the case of the eating action, for instance, the Carry Out rule removes the edible object from play — because, obviously, it has just been eaten.

Finally (unless an After rule has gotten in the way), the Report rule for the action will tell the player what has happened.

If we use the RULES debugging command to turn on reporting of rules, and then EAT something that we've declared as edible, we'll see exactly which rules are running:

```
>eat apple
[Rule "can't eat unless edible rule" applies.]
[Rule "can't eat clothing without removing it first rule" applies.]
[Rule "standard eating rule" applies.]
[Rule "standard report eating rule" applies.]
You eat the apple. Not bad.
```

It might seem odd at first that the "can't eat unless edible rule" and the "can't eat clothing without removing it first rule" apply — but in this situation, "apply" doesn't mean that the rules did anything. Plainly, the apple *is* edible, and it's not clothing. (The fact that Inform understands the concept of edible clothing is a little bizarre, but we won't worry about that.) All that's happening, during the processing of this command, is that Inform is *considering* those rules and then rejecting them. The result of the "standard eating rule" (a Carry Out rule) is that the apple is removed from the model world. It's gone, because the player has eaten it. The "standard report eating rule," a Report rule, then tells the player what has happened.

There are several ways for the author to get in and alter Inform's action processing. You've already seen a couple of examples. For more, see the section on "Altered Actions," below. Before we get to that, let's take a closer look at how action processing rulebooks work.

Rulebooks & "Stop the Action"

When a game is running, Inform processes actions using **rulebooks**. A rulebook is, as you might guess, a collection of rules. You can create your own rulebooks for any purpose at all if you want to, but that's an advanced programming concept, and it isn't covered in this *Handbook*. For many games, you won't need to. The rulebooks built into Inform can handle most types of action.

When your game is compiled, all of the Before rules — in your own code, in Inform's built-in library, and in any Extensions that you're using — are packed into the Before rulebook. Likewise, all of the Instead rules are packed into an Instead rulebook. (For now, we're not going to get into the question of how Inform decides what order to put the rules in. If you want to know more about this, you can read **Chapter 18** of the Documentation, "Rulebooks.") And all of the After rules are in a single After rulebook. But each action (for example, taking, looking, and going) has its own Check, Carry Out, and Report rulebooks.

Every rule in every rulebook ends in one of three ways: success, failure, or no decision. As an action is being processed, Inform runs through the rulebooks — Before, Instead, Check, Carry Out, After, and Report — consulting all of the rules in the order that they're listed in the rulebook. It goes through the rules *until one of the rules ends in success or failure*. At that point, Inform knows that the action has been handled, and it stops.

Every rulebook has a *default outcome* for the rules that are in the rulebook. If you write a new Instead, Before, After, Check, Carry Out, or Report rule and don't tell Inform what outcome your specific new rule will have, it will use the default outcome for that rulebook. This is important, because it affects how you write new rules.

The default for the Before, Check, and Carry Out rulebooks is to make no decision. So if you write a Before rule and don't tell Inform that the rule has succeeded or failed, Inform will look at your rule, do whatever it says to do, and then go right on processing the action. The After rulebook, however, has a default of success, and the Instead rulebook has a default of failure. So when you write a new Instead or After rule, if you don't say otherwise, your rule will cause action processing to stop.

The phrase "continue the action" is one of the first phrases I learned when I started writing new rules. It means exactly the same thing as "make no decision". You can use either of these two phrases. Here's a simple example:

```
Instead of taking the live wire:
	if the player does not wear the insulated gloves:
		end the game in death;
	otherwise:
		continue the action.
```

If the player is not wearing the insulated gloves, the outcome of the Instead rule will be failure (though in this case it hardly matters, as the game-ending rules will take over). If the player *is* wearing the gloves, this Instead rule will make no decision, and action processing will continue — most likely with the rules in the Check Taking rulebook. (Since the live wire is probably attached to something at its other end, and thus can't be carried around, this example doesn't actually make a lot of sense, but we'll ignore that.)

The opposite of "continue the action" is "stop the action". I've been in the habit of using this — but I'm going to have to get out of the habit. Most of the time, it works just fine, but once in a while it can get you in trouble. Let's look at why.

Here's a simple game, for testing, that shows what can happen. I like putting a red button in the room in a test game so that I can test anything I like using the command PUSH BUTTON. In a real game, the rule shown below for taking the sponge would be more complex; there's no need to ever do it as shown here. But the result (when the game is being played) might be similar to what you're seeing if you have a mysterious bug in your code.

```
The Lounge is a room.

A sponge is in the Lounge. A red button is in the Lounge.

Before taking the sponge:
	now the player carries the sponge;
	say "You grab the sponge.";
	rule succeeds.
	[stop the action.]
```

```
Instead of pushing the red button:
	if we have taken the sponge:
		say "Yep -- you took the sponge.";
	otherwise:
		say "Nope -- you never took the sponge."

test me with "take sponge / i / press red button"
```

For technical reasons, it’s necessary to test using the odd phrase “we have taken the sponge”. Testing “if the player has taken the sponge” won’t compile. But the point is this: If the Before rule (“Before taking the sponge”) ends with “rule succeeds”, as shown above, running the test by typing TEST ME will reveal that we have indeed taken the sponge. On the other hand, if you comment out “rule succeeds.” (by putting square brackets around it) and uncomment “stop the action.” (by removing the brackets), when you run TEST ME you’ll see that apparently we have *not* taken the sponge, even though we did.

That, in a nutshell, is why using “stop the action” is a bad idea. When you use “rule succeeds” or “rule fails” Inform does a little housekeeping behind the scenes. In this case, the housekeeping includes making note of the fact that the sponge has been taken. If you use “stop the action” the housekeeping never happens, so Inform’s record-keeping will include a mistake.

Still not convinced? Here’s another quick example.

```
The Lounge is a room.

Steve is a man in the Lounge.

A sponge is in the Lounge.

Persuasion rule for asking Steve to try taking the sponge:
	persuasion succeeds.

Instead of Steve taking the sponge:
	now Steve carries the sponge;
	now the unicorn is angry;
	say "Steve scoops up the sponge."
```

If all we want is for Steve to pick up the sponge, we don’t need an Instead rule. The Persuasion rule will cause him to carry out the action. In this example I’ve added a line about an imaginary unicorn, because it’s a quick example of why you might want to use an Instead rule rather than letting the standard library handle Steve’s action. The code seems very straightforward, but there’s a problem. Inform thinks that the action of Steve taking the sponge *failed,* because failure is the default outcome of the Instead rulebook. Here’s the output in the game:

```
>steve, take the sponge
Steve scoops up the sponge.

Steve is unable to do that.
```

Steve is now carrying the sponge, but the game has printed out an extra message giving the player bad information. To avoid this, we need to make sure the Instead rule succeeds:

```
Instead of Steve taking the sponge:
        now Steve carries the sponge;
        now the unicorn is angry;
        say "Steve scoops up the sponge.";
        rule succeeds.
```

When we add “rule succeeds”, Inform knows that the action succeeded, so no extra message will be printed out. These examples show that the action processing rulebooks have default outcomes, and that it’s important to know what the default outcomes are.

Three Types of Actions

We can separate actions into three main categories, depending on how many nouns (objects) the action applies to. Some actions take no nouns, some take one noun, and some take two. Another type of action take **topics** (that is, simple text) instead of a noun. (See “Actions with Topics” on p. 145.)

Commands like SLEEP and WAIT stand by themselves. It would make no sense at all to say SLEEP THE APPLE. Sleeping *on* or *in* something, such as a bed, would make sense, but the command SLEEP IN BED would trigger a different action (not the sleeping action but the sleeping in action), and you’d have to create that action yourself, because Inform doesn’t include it by default.

As this little digression should show, the same verb (in this case, “sleep”) can trigger two or even three different actions, depending on how the player’s input is phrased, and in particular depending on how many nouns are used in the command.

Commands like PICK UP THE APPLE and BOUNCE THE BALL take one noun. Commands like PUT APPLE IN BASKET and HIT BALL WITH STICK take two nouns. Two nouns are the upper limit; there’s no easy way in Inform to create a command that uses three nouns. Nor is there much reason to want to do so. The best advice I’ve read on this subject is, “If you think you actually need to create an action that requires three nouns … think about it some more.”

While processing the player’s command, Inform refers to the first noun in the input simply as the noun. The second noun is referred to as — you guessed it — the second noun. In writing our own code, we can test the values of noun and second noun if we need to, like this:

```
Instead of inserting the apple into something:
        if the second noun is the basket:
                say "The basket scoots away from you!";
        otherwise:
                continue the action.
```

In this case, we might be better off writing an Instead rule that simply refers to “inserting the apple into the basket". But you'll often find it useful to write tests such as, "if the second noun is the basket".

The second and third categories above use objects — what Inform calls *things*. But you can't refer to "the thing" or "the second thing" in processing an action. The word "noun" is the tool for the job.

Creating New Actions

To create a new action, we need to do three things:

1) We give the action itself a name, so that we'll be able to refer to it in our code.
2) We tell Inform exactly what forms of input will cause the action to happen.
3) We write some code that tells Inform exactly what to do when the action starts. (This would include both things that happen if the action succeeds, and messages that will print out if the action fails for some reason.)

Chapter 6 of the *Recipe Book,* "Commands," has a lot of good information on how to create your own actions. If you haven't read this yet, give it a look.

To illustrate the various possibilities for creating actions, we're going to create an action that will happen when the player uses the verb PAINT. Painting is not an action that's built into Inform, but painting something might be a way of solving a puzzle.

The most basic form of this action is one that takes no nouns at all — the bare command PAINT. We can create it this way:

```
The Test Lab is a room. "Many devious tests are conducted here. You can see a wooden fence."

A wooden fence is scenery in the Lab.

Painting is an action applying to nothing. Understand "paint" as painting.

Check painting:
	say "You don't have any paint." instead.
```

If you compile this game and test it, you'll see the following output:

```
>paint
You don't have any paint.

>paint the fence
I only understood you as far as wanting to paint.
```

The painting action as we've defined it does nothing at all: The check rule has no exceptions. We could, however, make an exception if we wanted to, using an Instead rule. For instance, we might do this:

```
Check painting when the location is the Paint Store:
	say "You find it impossible to choose from among the hundreds of tempting colors." instead.
```

Most often, we'd expect the player to want to paint some particular thing, so the example above is included mainly to illustrate the simplest way to create a new action, one that has no noun. We've defined the action ("Painting is an action applying to nothing.") We've told Inform what inputs will trigger the action ("Understand "paint" as painting.") And we've written a Check rule that will run when the painting action is being processed. Because we haven't defined any grammar that would match the command PAINT THE FENCE, the parser reports that it has failed: "I only understood you as far as wanting to paint."

It's important to remember that by default, a Check rule does *not* stop the processing of the action. After the Check rule runs, the Carry Out, After, and Report rules for the action will all run — unless one of them stops the process. To prevent this, if you want action processing to stop after your check rule, tack the word "instead" on at the end of the say phrase you're printing out in the Check rule. As you might guess, this causes the Check rule to operate more like an Instead rule, and Instead rules fail by default, so after your Check rule has run, action processing will halt.

Next, let's look at how to paint something. This example is going to get complicated, because in order to actually do any painting we'll need, at the very least, a paintbrush, and quite likely a can of paint as well. But we can start by creating the action we'll need. Remember, we've already created an action called painting (which applies to nothing), so we can't call our new action painting. Let's call it paint-applying:

```
Paint-applying is an action applying to one thing and requiring light. Understand "paint [something]", "put paint on [something]", and "apply paint to [something]" as paint-applying.

Check paint-applying:
	say "[The noun] look[if the noun is not plural-named]s[end if] fine the way [if the noun is plural-named]they are[otherwise]it is[end if]."
```

If you've included Plurality by Emily Short (recommended), the Check rule can be made a little simpler:

```
Check paint-applying:
	say "[The noun] look[s] fine the way [it-they] [is-are]."
```

As before, this action does nothing except print out an error message. But now PAINT THE FENCE will produce a sensible output: "The wooden fence looks fine the way it is."

The essential thing to notice is the use of the grammar token "[something]" in the action's Understand sentence. The token "[something]" will match any object that is in scope — that is, any object that is visible to the player character. This is the heart of creating new actions in Inform.

The phrase "and requiring light" in the definition of the action is useful mainly if your game includes any dark rooms. Painting is obviously an action that requires light, so if the player is in a dark room we want to make sure she can't paint anything, even if she does have all of the necessary painting implements.

The next thing we need to do, of course, is to add a painting implement, specifically a paintbrush, so that the player can type PAINT FENCE WITH PAINTBRUSH. For this tutorial, I'm not going to bother adding a paint can that the brush needs to be dipped into. This brush comes fully loaded with paint, which is not very realistic, but it will simplify the code in the examples.

When creating an action that requires two nouns, it's a very good idea to give the action a name that includes "it with", "it on", "it under", or some similar phrase. In this case, we're going to create an action called "painting it with". Giving the action this type of name — "xxxxing it with" — is not required, but if you forget to do it, your code will be hard to read, and you'll probably have more bugs to fix. In some cases, you might need to use a different preposition. For instance, you might need to create an action called "unfastening it from" or "placing it near".

Here is a fairly complete bunch of code that includes all three of our new painting actions. As you'll see, the Check rule has a fairly deep and tangled set of indentations. If you're trying out this game, you'll need to copy these with care. Even better would be to develop an understanding of how indentation works in Inform. For information on this, turn to p. 233.

```
The Test Lab is a room. "Many devious tests are conducted here. You can see a [wooden fence]."

A wooden fence is scenery in the Lab. The description is "It's a [if the fence is painted]freshly painted[otherwise]plain[end if] wooden fence." The wooden fence can be painted or unpainted. The wooden fence is unpainted. Understand "freshly" and "painted" as the wooden fence when the wooden fence is painted. The printed name of the wooden fence is "[if the fence is painted]freshly painted [end if]wooden fence".

A paintbrush is in the Lab. Understand "brush" as the paintbrush.

Painting is an action applying to nothing. Understand "paint" as painting.
Check painting:
    if the player carries the paintbrush:
        say "You'll need to be more specific about what you want to paint.";
    otherwise:
        say "You don't have any paint."

Paint-applying is an action applying to one thing and requiring light. Understand "paint [something]", "put paint on [something]", and "apply paint to [something]" as paint-applying.

Check paint-applying:
    if the noun is the fence:
        if the fence is painted:
            say "You already did that.";
        otherwise if the player carries the paintbrush:
            try painting the fence with the paintbrush instead;
        otherwise:
            say "You'll need to find a paintbrush somewhere if you want to do that.";
    otherwise:
        say "[The noun] look[if the noun is not plural-named]s[end if] fine the way [if the noun is plural-named]they are[otherwise]it is[end if]."
```

```
Painting it with is an action applying to two things and requiring light. Understand "paint [something] with [something]" as painting it with.

Check painting it with:
        if the second noun is not the paintbrush:
                say "[The second noun] can't be used for painting things." instead;
        otherwise if the player does not carry the paintbrush:
                say "You'd need to find a paintbrush somewhere if you want to do that." instead;
        otherwise if the noun is not the fence:
                say "[The noun] look[if the noun is not plural-named]s[end if] fine the way [if the noun is plural-named]they are[otherwise]it is[end if]." instead;
        otherwise if the fence is painted:
                say "You already did that." instead.

Carry out painting it with:
        now the fence is painted.

Report painting it with:
        say "You apply a fresh coat of paint to the fence."
```

This code is written in a way that assumes the player will never need to paint anything except the wooden fence. If several objects in the game can be painted, it would have to be changed. In that case, we might want to use Instead rules for each paintable object rather than Carry Out and Report rules. The code could be expanded in other ways. For instance, we might want to change the color of the fence after it's painted. But this example should give you a better idea how to create a new action.

You might notice a couple of other things too. We've allowed the fence to be either painted or unpainted. This is a new property we've created, which exists only for the fence object. This is so we can give the player an error message if he tries to paint the fence more than once. We've added some vocabulary words to the wooden fence object ("freshly" and "painted") that can only be used by the player after the fence is painted.

We've also told Inform that the printed name of the fence (the name printed out when the game runs) depends on whether it has been painted. Finally, the room description refers to "[wooden fence]", so the room description will also change after the fence is painted.

You may want to study the logic of the Check rule for painting it with. It's important to get the if-tests in a sensible order. The first question is, is the second noun something other than the paintbrush? If it's not — that is, if the second noun *is* the paintbrush — we then ask, has the player neglected to pick up the paintbrush? (It might be locked up in the glass case, or just lying on the ground.) Next, we check whether the noun is something other than the fence. If not — if the player is indeed trying to paint the fence — then the fourth question to ask is, is the fence painted already? Only if the answers to all four of those questions are "No" will the action processing move on to the Carry Out rule.

If these four questions are in some other order, the output might not be so sensible. If we put the question of whether the fence is painted first, for instance, this output could happen:

```
>paint the bucket with the apple
```

```
You already did that.
```

Trust me, players will sometimes try absurd commands like that, just to see what will happen. If they're beta-testing your game, that's what you *want* them to do. When creating a new action, it's a good idea to think about all of the silly things a player might try. What happens if she tries to perform the action on herself? On another character? On something she's wearing?

I tested the code above at least a dozen times while writing it. I tried all of the absurd inputs as I could think of, such as PAINT FENCE WITH APPLE, just to see what would happen. When writing a new action, you'll want to go through quite a bit of testing yourself before letting your trusted team of testers try torturing the game.

[Any Thing]

Earlier in this chapter, I told you that the first step in processing an action is to make sure the object or objects that the player is talking about are in scope — that is, that they're visible in the room. This happens because the grammar for actions normally uses the token "[something]". The code for the new action called paint-applying, a page or two back, gives a good example.

If we want to bypass scope checking, we can change this to the token "[any thing]". The token "[any thing]" puts every object in the game in scope, no matter what room it's in (or, for that matter, if it's nowhere). Please don't bypass scope checking for casual reasons, because it will make a mess of the realistic effects that Inform tries to create. Once in a while, though, it may be useful. For instance, you could give your players more or less the same power as the debugging PURLOIN command, like this:

```
Acquiring is an action applying to one visible thing. Understand "acquire [any thing]" as acquiring.

Check acquiring:
	if the player encloses the noun:
		say "You already have [the noun]." instead.

Carry out acquiring:
	now the player carries the noun.

Report acquiring:
	say "You are now carrying [the noun]."
```

Note the use of "one visible thing" in the definition of the acquiring action. When writing an action that can apply to "[any thing]", you need to do this. Oddly enough, "one visible thing" is a more *general* term than "one thing." In effect, "one thing" means "one thing that the player can both see and touch," while "one visible thing" means "one thing that the player can see (but we don't care whether the player can touch it or not)". If you fail to specify "one visible thing", your magic grab-anything command will fail: The parser will complain that the player is unable to reach into a distant room. In other words, the desired object will fail the touchability test.

The acquiring action might make a reasonable magic spell if your player character is a powerful wizard, able to summon distant objects, but you'll want to be careful how you write the Check rule for it. Several things can go wrong if you let the player use this type of spell indiscriminately. As mentioned in Chapter 5, "Characters" (see p. 162), the "[any thing]" token is also useful for creating topics of conversation — abstract objects that are never anywhere in the model world. But if you're doing this, and *also* using "[any thing]" to magically manipulate objects that do appear in the model world, you need to make sure the Check rule handles all of the possibilities. If you've created a SHAZAM spell that can acquire distant objects and "India" exists as an object that can be talked about, you don't want the naughty player to be able to do this:

```
> shazam india
With a swirl of sparkling light, India appears in your hands!
```

What's worse, SHAZAM ME will produce a run-time error.

For an example that might be more useful, let's suppose we want the player to be able to dig a hole. We'll give the player a shovel, and we'll assume the setting is outdoors. (If you have both indoor and outdoor rooms, you'll need to do some extra checking before you let the player dig a hole!) The problem we have to solve is that the hole isn't in scope until after the action takes place. There isn't any hole in the room until the player digs it, so how can we arrange matters so that the player can use the command DIG HOLE?

The way to solve this little problem is to use "[any thing]". We'll create the hole offstage and then move it into the room when the player gives the command. Here's a simple game that shows how to do this:

```
The Forest Path is a room. "Tall trees surround you."

The player carries a shovel.

The hole is an open container. The hole is fixed in place. [The hole starts out nowhere.]

Rule for writing a paragraph about the hole:
	say "You've dug a hole here."

Digging is an action applying to one visible thing. Understand "dig [any thing]" as digging.

Check digging:
	if the noun is not the hole:
		say "[The noun] can't be excavated." instead;
	otherwise if the holder of the hole is a room:
		say "You've already dug a hole." instead;
	otherwise if the player does not carry the shovel:
		say "You scrabble around with your hands, but the dirt is pretty hard. You need a
shovel." instead.

Carry out digging:
	try digging the hole with the shovel instead.

Digging it with is an action applying to one visible thing and one thing. Understand "dig [any thing] with
```

```
[something]" as digging it with.

Check digging it with:
	if the noun is not the hole:
		say "[The noun] can't be excavated." instead;
	if the second noun is not the shovel:
		say "[The second noun] can't be used for excavating." instead;
	otherwise if the holder of the hole is a room:
		say "You've already dug a hole." instead;
	otherwise if the player does not carry the shovel:
		say "You can't do that unless you're actually holding the shovel." instead.

Carry out digging it with:
	move the hole to the location.

Report digging it with:
	say "You pitch in with the shovel and dig a nice deep hole."
```

Same Action, New Results

Some puzzles are designed to force the player to take a given action several times. For instance, the player might be required to knock on a door three times before someone will open the door. To do this in Inform, you would normally use Instead rules and the phrases “for the first time,” “for the second time,” “for the third time,” and so on. It’s important also to write a default rule for handling the same action. In the example below, the Report rule for our new knocking on action takes care of this.

```
The chapel door is west of the Grand Entry Hall and east of the Chapel. The chapel door is a door.
The chapel door is scenery and locked.

Knocking on is an action applying to one thing and requiring light. Understand "knock on [something]"
and "tap on [something]" as knocking on.

Check knocking on:
	if the noun is the player:
		say "Your head makes a hollow sound." instead;
	otherwise if the noun is a person:
		say "That wouldn't be polite." instead.

Report knocking on:
	say "You tap gently on [the noun], but nothing happens."

Instead of knocking on the chapel door for the first time:
	say "The sound of your knuckles generates booming echoes within the chapel. For a moment
you think you hear someone moving around inside, but the door remains closed."

Instead of knocking on the chapel door for the second time:
	say "Within the chapel, a faint and uncertain voice cries, 'I'm coming, I'm coming.' You wait
patiently for a minute, but the door doesn't open."
```

```
Instead of knocking on the chapel door for the third time:
	now the chapel door is open;
	say "After a few more interminable minutes of waiting, the door swings ponderously open. A man in a clerical collar, who looks to be at least a hundred years old, peers out at you through rheumy eyes. 'I'm sorry,' he says. 'We're no longer admitting worshippers. God is dead, you see.'"
```

If you write a puzzle like this, it's important to give the player a clue that the action should be repeated. If the first response to KNOCK ON DOOR is a bare "Nothing happens" or some similar phrase, the player is unlikely to try the action again.

Redirecting an Action

Situations can arise in which a game needs to respond to a particular command by turning it into a different command. The tool for this is, as you might guess, the Instead rule. When the player tries one command, an Instead rule intercepts it and turns it into a different command.

Here's a simple example. (For a slightly more complex example, see "Mr Potato Head" in Appendix B, p. 265.) The player is in a room called At the Foot of a Tree. Perched in the Tree is a separate room above it. The tree itself is scenery in the first room, and we want the player to be able to climb the tree. The command CLIMB TREE will result in the same action as the command GO UP:

```
At the Foot of a Tree is a room. "You're standing at the foot of a tall tree. Sturdy low-hanging branches suggest that it may be easy to climb."

The tall tree is scenery in At the Foot of a Tree. Understand "sturdy" and "branches" as the tall tree.

Perched in the Tree is up from At the Foot of a Tree. "The view from up here is spectacular (though admittedly rather leafy)."

Instead of climbing the tall tree:
	try going up.
```

It may seem that we could just as easily have said, "Instead of climbing the tall tree: now the player is in Perched in the Tree." And indeed, in the example above, that would produce pretty much the same result (although there would be a missing empty line in the output between the player's command and the room name). But redirecting the CLIMB TREE action to the GO UP action allows us to handle more complex game situations in a graceful way.

Let's suppose, for instance, that the player may be carrying something so heavy that it would make climbing the tree difficult or impossible. If we've redirected CLIMB TREE to GO UP, we only need to test this condition once. If CLIMB TREE and GO UP remain separate actions that send the player to the same destination, then we'll have to test what the player is carrying in two different pieces of code, which makes the code harder to write and harder to debug.

Because the Instead rule shown above redirects the climbing action to the going up action, turning the tree-climbing action into a puzzle (absurdly easy, but a puzzle for all that) requires very little effort.

After the code above, we add the following:

```
The player carries a heavy iron anvil. The description is "The iron anvil must weigh at least a hundred pounds."

Instead of going up in At the Foot of a Tree:
	if the player encloses the anvil:
		say "You'll never be able to climb the tree while carrying something so heavy.";
	otherwise:
		continue the action.
```

Now the player will see the same “you can’t do that” message, whether he typed CLIMB TREE or UP.

Inform allows you to test what action the player is attempting, but the syntax is a little tricky. For an example that shows how to do it, see “Restraints” on p. 272.

Actions with Topics

Inform provides an action called consulting it about. By default, this does nothing except print out a message that says, “You discover nothing of interest in [the noun].” The results, if the player tries LOOK UP EXPLOSIVES IN MRS SMITH, are rather comical, because Inform seems to be implying that there is actually a way of looking things up in Mrs. Smith. We’ll want to fix that. Nonetheless, we can use this action to create an encyclopedia — or for that matter, a computer terminal — in which the player can look up whatever topics we like. Here’s an encyclopedia:

```
Include Plurality by Emily Short.

The encyclopedia is in the Library. The description is "A massive 27-volume set of the Encyclopedia Frobozzica. You could probably look up almost anything in it!" Understand "volume" and "book" as the encyclopedia.

Instead of consulting the encyclopedia about something:
	say "Flipping through the encyclopedia, you learn that [run paragraph on]";
	if the topic understood is a topic listed in the Table of Encyclopedia Entries:
		say "[article entry][paragraph break]";
	otherwise:
		say "there's nothing in the encyclopedia on that topic."

Table of Encyclopedia Entries
topic	article
"Monty Python" or "Python" or "Monty Python's Flying Circus" or "Flying Circus"	"Monty Python's Flying Circus was a very strange British comedy show popular in the 1970s."
"crayons" or "crayolas" or "crayon"	"crayons are made of wax. They come in bright colors, and are used to create works of art on paper."
"weapons" or "weaponry" or "swords" or "sword"	"a sword was the weapon preferred by knights in the Middle Ages. Swords are often used for hacking, slashing, and stabbing."

Instead of consulting someone about something:
	say "Does [the noun] look like an encyclopedia?"

Instead of consulting something about something:
	say "[The noun] [is-are] not a likely-looking source of information."
```

The main part of this code uses a *table*. Tables are Inform's way of keeping large amounts of data organized; for more on tables, see p. 241. The main thing you need to know about tables, at the moment, is that they contain rows and columns, and that the items in each row are separated by Tab characters. There is a single return character at the end of each row. ***Note:*** Tab characters are *not* kept by a copy-paste action if you select the code above in Adobe Reader (Windows) or Preview (Mac) and copy it into Inform. If you do that, you'll need to replace the Tabs yourself, in order to get the game to compile. In addition, extra return characters may be *added* when you copy and paste text from the *Handbook.* These returns will have to be stripped out after you copy the example into Inform's IDE.

Inform's IDE doesn't display Tab-separated tables any better than the word processor I'm using to write this book. As a result, a table that has long entries, like the one above, can be difficult to understand or debug just by looking at it. Here's what the table would look like if we could use a real word processor table (which we can't do, because Inform wouldn't understand it):

topic	article
"Monty Python" or "Python" or "Monty Python's Flying Circus" or "Flying Circus"	"Monty Python's Flying Circus was a very strange British comedy show popular in the 1970s."
"crayons" or "crayolas" or "crayon"	"crayons are made of wax. They come in bright colors, and are used to create works of art on paper."
"weapons" or "weaponry" or "swords" or "sword"	"a sword was the weapon preferred by knights in the Middle Ages. Swords are often used for hacking, slashing, and stabbing."

With the code shown above, the player can CONSULT ENCYCLOPEDIA ABOUT SWORDS or LOOK UP SWORDS IN ENCYCLOPEDIA. Both commands lead to the consulting it about action. If you look at the entries in the topic column for the table, you'll find that each entry has several texts separated by the word *or.* It's important to spell out all of the variations you think your player might try to use. The last row in the table, for instance, has no entry under topic for "weapon", because I forgot to add one. If the player tries to LOOK UP WEAPON IN ENCYCLOPEDIA, the default response ("...there's nothing in the encyclopedia on that topic") will be printed out.

The last two Instead rules in this example provide better responses for when the player tries to look up a topic in something other than the encyclopedia.

Action Processing — Summary

The table below (which was suggested by Michael Callaghan) shows all of the stages Inform goes through when processing a player's command — or rather, all of the stages Inform normally goes through. The action processing can be stopped at every stage. If it's stopped at a given stage, none of the later stages will be reached.

Stage	Comment	Outcome	How to change
Parsing	The player's input is checked to make sure it makes sense — that all of the words are spelled correctly and so on.	If the input makes sense, Inform 7 next goes on to check **Scope.** If the input doesn't make sense, an error message is displayed.	It's possible to modify how the parser understands the player's input, but this is an advanced topic not covered in the *Handbook*.
Scope	Where the action involves one or more things, Inform 7 checks that the things referred to are in scope. With some exceptions, things that are in scope are in the same room as the player.	If the things are in scope, Inform 7 checks the **Before rules.** If the things are not in scope, an error message is displayed.	There are two ways in which scope can be changed. The first, discussed in this chapter, is where the action affects "[any thing]." The second is to write some special code that will place something in scope when it would otherwise not be. This is an advanced topic not covered in the *Handbook*.
Before rules	All Before rules are contained in a single rulebook that Inform consults to see if any of them applies to the action that is now being processed. Before rules are useful where you want to carry out an action before Inform 7 goes on to the next two stages **(touchability** and **implicit** actions).	The default outcome after consulting the Before rules is to make no decision, so that Inform 7 proceeds to consider **touchability**. If any of the Before rules being considered by Inform specifies that the action has succeeded or failed, Inform stops processing the action.	To indicate that a Before rule has succeeded (in doing something) and stop processing the action, use "rule succeeds". To indicate that a Before rule has failed and stop processing the action, you can use "rule fails" or "instead".
Accessibility	Inform checks that even if something affected by the action is in scope (see above), it needs to be visible and touchable if the action involves touching it. If an object is in a closed glass container, for instance, the player will be able to EXAMINE it but not TAKE it.	If the thing cannot be touched, if it's hidden, or if the room is dark, an error message is displayed. If the thing can be touched, Inform then goes on to consider any implicit actions that may be called for.	The easiest way to bypass the accessibility test is to intervene using a Before rule.
Implicit actions	Certain verbs trigger an implicit action. For example, the EAT command causes an implicit taking action if the	If the implicit action fails, an error message is displayed.	If you don't want the implicit actions to be triggered, the easiest way to do this is to intervene using a Before rule.

	item to be eaten is not being held by the player.	If the implicit action succeeds, Inform goes on to consider any **Instead** rules that may apply.	
Instead	All Instead rules are contained in a single rulebook that Inform consults to see if any of them applies to the action being carried out. Instead rules are the most useful rules for changing the results of actions (both those in the standard rules and those you've created in your own source code).	The default outcome for the Instead rulebook is failure. If any instead rule applies, the action processing stops with the outcome "rule failed". If no Instead rules apply, Inform next considers the **Check** rules.	If we want an Instead rule to stop the action with the result that the action succeeded, we can use the words "rule succeeds". If we want action processing to continue, we can use the words "continue the action".
Check rules	Every action has its own Check rulebook. If you are creating new actions, this is where you will set out any preconditions that apply to the action being carried out. For simple actions, it is possible to include all of the actions rules in one Check rule.	The default outcome from the Check rules is to make no decision and proceed to consider the **Carry out** rules.	To specify that a Check rule has succeeded and stop processing the action, you can use "rule succeeds". To specify that a Check rule has failed and stop processing the action, you can use "rule fails" or "instead".
Carry Out rules	Every action has its own Carry Out rulebook. If you're creating new actions, you'll write a Carry Out rule that handles what's supposed to happen.	The default outcome from the Carry Out rules is to make no decision and proceed to consider the **After** rules.	Normally, you should not need to change the default outcome from the Carry Out rules. To specify that a Carry Out rule has succeeded and stop processing the action, you can use "rule succeeds". Once the Carry Out rules have been reached, the action is considered by Inform to have succeeded, so it make no sense to use "rule fails" or "instead" within a carry-out rule.
After rules	All After rules are contained in a single rulebook that Inform consults to see if any of them applies to the action being carried out.	The default outcome for the After rules is success. If an After rule applies, Inform stops processing the action.	You won't often need to change the default outcome of an After rule. But if you want Inform to go on to the Report rule, end your After rule with "continue

	After rules are useful where you want to make other changes to the world after the action has succeeded.	If none of the After rules applies to the action, Inform proceeds to the final stage of action processing and considers the **Report** rules.	the action”.
Report rules	Every action has its own Report rulebook. When you create new actions, write a Report rule to print out the messages that will be displayed to the player once the action has succeeded.	The default outcome for the Report rules is success.	As this is the last stage of processing the action, it is not appropriate to use “rule fails”, “rule succeeds” or “instead”. In rare cases, you may have two Report rules that both apply to one action. You can use “continue the action” in the first of the rules so that the second Report rule will run.

Chapter 5: Creating Characters

Stories are about people. Because interactive fiction is a form of storytelling, it's almost inevitable that you'll want to include a few people in your game. A story with no people in it probably won't be very interesting or fun — at least not for very long. If your players can't do anything but wander around picking up treasures and fighting off monsters, before too long they'll start to get bored. Besides, even a monster is a person, loosely speaking.

Using Inform, you can add people (or talking animals, or whatever sort of odd characters you'd like) to the games you write. Creating lifelike characters is an important skill to learn, and it may be the most complicated part of writing IF. In this chapter we'll cover the basics of creating characters and also look at a few advanced techniques that you'll probably want to use as your story develops.

Programming a non-player character (the abbreviation "NPC" is used throughout the interactive fiction community to refer to a non-player character) is complicated because people are complicated. Think about the differences between interacting with your Uncle Fred and interacting with a bowling ball. Uncle Fred does things on his own — that is, he initiates activities. A bowling ball doesn't. Uncle Fred may observe you and comment on something you're doing. A bowling ball won't. Uncle Fred may respond to you differently depending on whether he's happy or sad, or on whether he's upset with you. A bowling ball will respond exactly the same whether it's red or black or zebra-striped.

The ways that people interact with one another are also complicated: Uncle Fred might respond differently if you shout or whisper, but a bowling ball won't. To make matters worse, shouting and whispering are not possible in most IF conversation systems.

Whoever plays your game will naturally hope (or worse, expect) that the player character will be able to interact with other characters the same way the player herself would interact with a real person. Sadly, that degree of realism is just about impossible to achieve using today's IF programming tools. The parser is designed to accept in puts in the form <VERB> <NOUN> <PREPOSITION> <NOUN>. Anything more complicated is not likely to work well.

Why not design a better parser, then? This turns out to be a very difficult problem in computer science. Various people have tried it, but so far, the results have not been too inspiring. Technically, it’s possible to write a game that will respond to whatever complicated player inputs you’d like. If you want your game to respond to ASK EMILY WHETHER EMILE’S DOG IS STILL TOO SICK TO HUNT FOR THE SQUIRREL, you can do it using the “after reading a command” activity, as explained on **p. 17.31** of the Documentation, “Reading a command.” But there are several problems with this type of trick, including the problem of letting the player know that such a complex input is possible. I don’t recommend that you go down that road, especially not in your first game.

Believable Characters

Characters will seem more believable if the player’s interaction with them is limited. This could be because the NPC plays a limited role in the story, or does not know the PC (player character): It’s a lot easier to make a guard believable than a girlfriend! You can also limit the interaction with a character by moving the character onstage and offstage again quickly. Characters who have a single-minded obsession with one thing (for instance, a king who only cares about whether you’re a spy) are easier to create than characters who are just hanging out.

Even while sticking within the limitations of “normal” IF programming, though, we can employ various tricks to make our characters seem more real. The characters may be just the software equivalent of cardboard, but players will enjoy meeting and talking with them.

Before we start talking about specific programming techniques, we need to talk about how you might want to use characters in your interactive story. What roles or functions might they take on?

Overview

To begin with, your players will probably want to be able to talk to the other characters. There are several ways to set up a conversation system. Each method has advantages, and also weaknesses, as we’ll see.

One common reason to want to have a conversation with a character is to try to learn something useful. If you’re writing a game about criminals, for instance, ASK POLICEMAN ABOUT BURGLAR might be an important command for the player to try. Another reason to have a conversation is because you hope to cause the character to do something. If the player types TELL POLICEMAN ABOUT THE BURGLAR, the policeman might rush off to arrest the burglar — a real change in the world of the story that won’t take place unless the player gives that command.

The player character may need help doing something, and may want to ask another character for help. For instance, a player character who has only normal strength might want to enlist the aid of a weight-lifter to move a boulder. So the player may need a way to give instructions to other characters. ASK REGINALD TO LIFT THE BOULDER could be an important part of the game.

Characters may have possessions that the player needs. One common type of instruction is to ask for an object. If you ASK MARY FOR DIAMOND NECKLACE, she might refuse to part with it, or she might give it to you, depending on how the game is written.

Some characters are stationary — they're always in one room. Other characters may wander around the map. They may follow the PC, or run away when the PC approaches.

Characters will sometimes carry out simple tasks on their own, such as stealing things from the player or even killing the player.

When a character is in the same room as the player, the character may seem more lifelike if it's carrying out some sort of "stage business," engaging in a minor activity every turn, or once every few turns, just to remind the player that there's someone else in the room. If the character is doing this type of stage business, nothing is likely to change in the room (although it could — the butler might be polishing the silverware for 20 turns, and at the end of that time the description of the silverware might change to say that it's spotless and shining). All that usually happens is that the game prints out a brief message describing a nonexistent action.

If your story takes place in a world where there's lots of fighting (swordfights or laser guns, your choice!), you may want the player to engage in hand-to-hand combat with other characters. We'll take a brief look at how to set this up.

Groups of characters require special handling. If your player character is facing a crowd of angry villagers armed with torches and pitchforks, you probably don't want to create a hundred different villager characters. For one thing, it's a lot of work, and you won't gain much in terms of realism or a more dramatic story. A better method will usually be to create a single "crowd" object (which might not be a person at all, as far as Inform is concerned) and then create a single "spokesperson" man or woman, an actual Inform person, who will engage in any actual conversation or other activities. On the other hand, if you're writing an Agatha Christie type of murder mystery, you may want to have as many as six or seven individual characters (the suspects) all sitting around in the drawing room at the same time. Inform provides some fascinating tools with which to allow these characters to relate to one another. (See the discussion of relations in Chapter 9 of this *Handbook,* on p. 253.)

The Player Character

Pleasantly little needs to be said about creating the player character (PC). Your PC can have any identity you might like, from a mile-high evil robot to a small, cuddly bunny rabbit. Or the PC could be a sort of blank — a shadow character that the actual player can project his or her own personality into.

To start creating a PC, all you need to do is write a description:

```
The description of the player is "In the absence of a mirror, you're not quite sure what you look like today."
```

In games that are puzzle-oriented rather than people-oriented, that may be all you need to do. Depending on the character you've chosen, though, you may want to write non-standard messages for situations in which the PC either does something, or doesn't. For instance, your PC might be very timid. In that case, the PC would naturally hesitate to do certain things that another PC might have no trouble doing:

```
Instead of taking the spider, say "You can't bring yourself even to get near it."
```

At the other extreme, the PC might be rambunctious, ill-mannered, or just plain evil. He might enjoy breaking things, for instance. This fact could lead you to write code like this:

```
Some dishes are in the Lab. The dishes can be broken or unbroken. The dishes are unbroken. The printed name of the dishes is "[if broken]broken [end if]dishes". Understand "broken" as the dishes when the dishes are broken.

Instead of attacking the dishes:
	if the dishes are broken:
		say "You already took care of that little chore.";
	otherwise:
		now the dishes are broken;
		say "You smash the dishes with gusto!"
```

The point of this code is to respond to the command BREAK THE DISHES with the line, "You smash the dishes with gusto!" This is a way of giving a specific character to the PC.

In the early days of interactive fiction, a number of games were released in which the PC changed identity during the course of the game — essentially, switching from one body to another. This isn't done too often anymore, but if you want to do it, it isn't difficult. All you need is a couple of lines of code. In the silly little example below, we'll use pressing a button to cause the PC to swap bodies with an NPC. In a real game, you might use a magical device of some sort, or have the PC wake up from a dream and discover that she's really someone else.

```
The Test Lab is a room. "Many devious tests are conducted here."

Bingo is an animal in the Lab. The description is "[if the player is Bingo]You are[otherwise]Bingo is[end if] a peppy-looking cocker spaniel."

Bob is a man in the Lab. The description is "[if the player is Bob]You are[otherwise]Bob is[end if] a heavyset, muscular man."

The player is Bob.

Linda is a woman in the Lab. The description is "[if the player is Linda]You are[otherwise]Linda is[end if] a sultry and curvaceous beauty."

The red button is in the Lab.

Instead of pushing the red button:
	if the player is Bob:
		now the player is Linda;
		say "A strange tingly feeling overtakes you. You feel much more feminine than before.";
	otherwise if the player is Linda:
		now the player is Bingo;
		say "Your head spins. When you recover, things seem to have changed. The world is
larger, and you're covered with fur.";
	otherwise:
		now the player is Bob;
		say "You feel odd for a moment Yes, you're human again, and muscular, and male."
```

In a real game, many other messages and rules would have to test whether the player is Bob, Linda, or Bingo, and change the results of the player’s commands based on the PC’s identity.

Creating an NPC

Creating a new non-player character is easy. Assuming we’ve created a room called the Billiard Room, we can then write:

```
Troy is a man in the Billiard Room. The description is “Troy looks devilishly handsome in his tuxedo, but you’re not sure you trust him.” Troy is wearing a tuxedo.
```

The assertion, “Troy is wearing a tuxedo” both creates an object (the tuxedo) and lets Inform know that the tuxedo is wearable. (If you actually want the player to wear the tuxedo at some point in the story, you’ll have to write some code for getting it away from Troy. By default, characters won’t let go of things they’re wearing or carrying.)

In fact, the sentence “Troy is wearing a tuxedo” lets Inform know that Troy is a person, because only people and animals can wear things. By default, an object that you say is wearing something will be male and a person, not an animal. So we can delete the words “a man” in the code above and get exactly the same result. I suggest always saying that an NPC is a man or a woman, however, because if you should later change your mind about having the NPC wear something and delete the sentence about the tuxedo, the NPC will no longer be a man, just a thing.

If you use this code in a game, you’ll find that when the player enters the billiard room, the game reports, “You can see Troy here.” As you’ll recall from Chapter 3, we can improve the output by giving the Troy object an initial appearance. The initial appearance is formatted exactly like a room description — it appears immediately after we create Troy, and is written as a double-quoted sentence that stands by itself. The sentence below that begins, “Troy is leaning nonchalantly” is an initial appearance.

```
Troy is a man in the Billiard Room. “Troy is leaning nonchalantly against a corner of the billiard table, pretending not to notice you.” The description is “Troy looks devilishly handsome in his tuxedo, but you’re not sure you trust him.” Troy is wearing a tuxedo.
```

Now, when the player enters the billiard room, the room description will include, instead of the boring line “You can see Troy here,” the much more colorful line, “Troy is leaning nonchalantly against a corner of the billiard table, pretending not to notice you." I'd recommend always giving an NPC an initial appearance.

At present, Troy doesn't do much. But even at this stage, Inform understands that a person is different from most other types of objects. Here's a transcript of what can happen in the billiard room with no more code for Troy than what we've written above:

```
>kiss troy
Keep your mind on the game.

>kiss tuxedo
You can only do that to something animate.

>take troy
I don't suppose Troy would care for that.

>take tuxedo
That seems to belong to Troy.
```

As you can see, Inform automatically understands that kissing is something that the player character can do to people (or to animals), but not to inanimate objects. If we try to take a person, the default error message prevents it — and if we try to take something that a person is wearing or carrying, we get a different error message.

Incidentally, if you need the player to be able to kiss an inanimate object — perhaps you've created the Pope as a character, and the player is expected to kiss the Pope's ring — you'll find that a simple Instead rule won't do the job. A Before rule won't work either. This is because the Inform library defines the kissing action more or less like this:

```
Understand "kiss [someone]" as kissing.
```

The grammar token "[someone]" won't match an inanimate object; it will only match a person. So an Instead or Before rule will never have a chance to get into the act: the command KISS RING will be rejected by the game's parser. The same thing happens, by the way, with the GIVE TO and SHOW TO actions, though those are less likely to cause problems, because it's hard to think of a reason why you would want the player to be able to show or give something to an inanimate object!

We'll borrow a bit from Chapter 4 of the *Handbook*, "Actions," and show how to make it possible to kiss something inanimate. It's even easier than the method suggested in version 1 of the *Handbook*. We don't need to define a separate kissing action for inanimate things — all we need to do is add a grammar line and define a default "you can't do that" message:

```
Understand "kiss [something]" as kissing.
Instead of kissing something which is not a person.
	say "[The noun] do[if the noun is not plural-named]es[end if]n't look very sanitary."
```

Now we can write an instead rule that will allow the player to pay oscular obeisance to Troy's tuxedo:

```
Instead of kissing the tuxedo:
	say "Troy graciously allows you to kiss his tuxedo."
```

The distinction between “[someone]” and “[something]” is worth mentioning because it allows us to write new actions that will also apply only to people. For instance, we might want the player to be able to flatter other characters. Causing the flattery to affect the character would take a bit more code, but we can create a basic flattering action like this:

```
Flattering is an action applying to one thing. Understand "flatter [someone]" and "praise [someone]" as flattering.
Check flattering:
	say "[The noun] blushes modestly."
Instead of flattering Troy:
	say "Troy grins at you. 'Well, sure. Anything nice you say about me, I gotta believe you mean it.'"
```

Now the player can flatter any character in the game, which will produce the response in the Check rule above. But flattering Troy will produce a different response, and the parser won’t let the player flatter inanimate objects. Care is required when writing default responses, however, because an animal will match the grammar token “[someone]”. If you’ve got a talking tortoise in your game, you don’t want the tortoise blushing modestly when flattered!

“Man” & “Woman”

If you create an NPC by saying, for instance, “Troy is a man,” the parser won’t understand the word “man” as referring to Troy. This is because the name of a kind is not understood as referring to specific objects of that kind unless you say it is. You have to add the vocabulary by hand:

```
Troy is a man in the Billiard Room. Understand "man" as Troy.
```

If you have several male characters in your game, you can handle this for all of them at once by writing:

```
Understand "man" as a man.
```

Mr. & Mrs.

In the U.S., we always put a period after abbreviations like Mr., Mrs., Ms., and Dr. In Britain no periods are used. But even if Graham Nelson weren’t British, the period has some uses in Inform that would make it tricky to handle those abbreviations.

In source text, Inform understands the period as being the end of a sentence. So you can’t create a character this way:

```
Mrs. Smith is a woman in the drawing room. [Error!]
```

Inform will think “Mrs” is a separate sentence — a sentence that makes no sense. The solution is to leave out the period when creating Mrs. Smith:

```
Mrs Smith is a woman in the Drawing Room. "Mrs. Smith is a stout woman of mature years."
Understand "stout" and "woman" as Mrs Smith.
```

You can refer to her as “Mrs. Smith” in the description, or in any other output text that you write, but as far as Inform is concerned, she’s Mrs Smith. To fix this, we need a special rule:

```
Rule for printing the name of Mrs Smith: say "Mrs. Smith".
```

One problem remains: If the player types X MRS. SMITH, the parser won’t know what the player means. Inform won’t allow periods to be used in Understand text, so we can’t do this:

```
Understand "stout", "woman", and "Mrs." as Mrs Smith. [Error!]
```

Also, if the player types a command that includes a period, the parser will understand it as two separate commands. For example, the player can move quickly from room to room (assuming she knows the route) like this:

```
>n. e. n. nw
```

As far as the parser is concerned, those are four separate commands. This is a handy feature found in most modern IF interpreters. In the example above, X MRS. SMITH would produce the description of an object whose name includes the word “Mrs”, because the parser would understand the first command as X MRS — but SMITH would look to the parser like a separate command, one that is not likely to be understood. To prevent this, we have to use two advanced features of Inform — the “after reading a command” activity, and **indexed text**. Indexed text is sort of like regular quoted text, but the author can do things with it more easily. Here, we’re going to find an input that matches “MRS.” and replace it with “MRS” before the parser gets a chance to process it. While we’re at it, we’ll fix the other common abbreviations too:

```
After reading a command:
        let T be indexed text;
        let T be the player's command;
        replace the regular expression "mrs\." in T with "mrs";
        replace the regular expression "mr\." in T with "mr";
        replace the regular expression "ms\." in T with "ms";
        replace the regular expression "dr\." in T with "dr";
        change the text of the player's command to T.
```

I’m not going to try to explain every line in that code, because indexed text is an advanced topic, not something we’ll explore in detail in the *Handbook*. If you’re curious, you can learn more by reading **Chapter 19** of the Documentation, “Advanced Text.” If you don’t have the patience for that, just copy the code above exactly, and it will do the job. (If you’re retyping it, don’t skip the backslash characters before the periods. The backslash may look like a typo, but it belongs there.)

The Extension called Punctuation Removal by Emily Short provides a simple way of getting rid of the periods in the player's input.

Conversations, Part I: talk to

The easiest way to let your players talk to NPCs is by creating a "talk to" action. Here's a code sample that will produce a simple conversation:

```
Talking to is an action applying to one visible thing.
Understand "talk to [someone]" or "converse with
[someone]" as talking to.

Check talking to: say "[The noun] doesn't reply."

Instead of talking to Troy:
	say "[one of]'Hi, there,' you say confidently.[paragraph break]'What's happening?' he replies casually.[or]'I've been meaning to ask you about that tuxedo,' you comment. 'Where did you get it?'[paragraph break]'My tailor is quite exclusive,' Troy replies, inspecting his cuff. 'He would never consent to clothe riffraff like you.'[or]'You really are a stuck-up snob, aren't you?' you say hotly.[paragraph break]Troy laughs heartily. 'I was just yanking your chain. I bought it at Macy's for $60 at a clearance sale. I'll give it to you if you like.'[or]You decide against talking any further with Troy right now.[stopping]".
```

The most interesting thing about this code is what happens in the long "say" block. This is set up to give Inform some alternative outputs. The structure looks like this: "[one of]...[or]...[or]...[or]...[stopping]". This structure is covered on **p. 5. 6** of the Documentation, "Text with random alternatives"; also see Chapter 8 of the *Handbook,* p. 212.

When you use this type of code, the player can type TALK TO TROY, read the first response (up to "[or]"), then type AGAIN (or just G) to read the second response, and so on. The last response (the one just before "[stopping]" should always be used to indicate to the player that there's nothing further to be gained by trying to talk to the character. The output will look like this:

```
>talk to troy
"Hi, there," you say confidently.

"What's happening?" he replies casually.

>g
"I've been meaning to ask you about that tuxedo," you comment. "Where did
you get it?"

"My tailor is quite exclusive," Troy replies, inspecting his cuff. "He
would never consent to clothe riffraff like you."
```

```
>g
"You really are a stuck-up snob, aren't you?" you say hotly.

Troy laughs heartily. "I was just yanking your chain. I bought it at Macy's
for $60 at a clearance sale. I'll give it to you if you like."

>g
You decide against talking any further with Troy right now.
```

The good thing about a talk to conversation system is that it's easy to set up. The bad thing is that it's quite limited. If the player wants to ask or tell the character about three or four different things that may be significant to the story, a talk to system won't really do the job. Another problem is that if the conversation is going to move the story forward, it's up to the player to use the AGAIN command several times in order to make sure the conversation is finished.

Yet a third problem is that if the player should try TALK TO TROY only twice, go to a different room, return a hundred turns later, and try TALK TO TROY again, the conversation will continue as if there had been no interruption. This is not realistic.

One easy way to make TALK TO work more flexibly uses Inform's scenes feature (see Chapter 7 of the *Handbook*):

```
Instead of talking to Troy during Cocktail Hour:
```

A different but also handy use for the TALK TO command might be to have your game print out some suggested topics for conversation:

```
>talk to bishop
You could ask the bishop about the crypt, about the impending eclipse, or
about the Uzi he's carrying.
```

This list of possible topics can be fixed, or you can write some code that will put it together while the game is being played. After the player has asked about the crypt, for instance, this topic might disappear from the list of suggested topics, that topic being exhausted. An Extension called Conversation Suggestions by Eric Eve will give you a framework for setting up topic suggestion lists of this sort.

Conversations, Part II: ask/tell/give/show

Personally, I favor the ask/tell/give/show conversation system. (Other game authors disagree.) I feel it gives a good compromise between realism and challenging the player to figure out what's important and therefore worth having a conversation about.

This system is based on five commands:

ASK NPC ABOUT X

TELL NPC ABOUT X
SHOW X TO NPC
GIVE X TO NPC
ASK NPC FOR X

The give to, show to, and ask for actions are included in Inform. To use them in their basic form, all you need to do is write a few Instead rules. The ask about and tell about actions are included in Inform in a limited way: You can ask about or tell about topics (which are just strings of text), but not about objects that exist in the game unless you set up topics that have the same vocabulary as the objects. We'll take a close look at all of these commands, and show ways to make them more flexible.

The simplest command to add to your game is give to. Continuing the earlier example game, with Troy in the Billiard Room, we can do this:

```
The player carries a plum.

In the Billiard Room is a hammer.

Instead of giving the plum to Troy:
        say "Troy inspects the plum carefully, accepts it, and pops it into his mouth.";
        remove the plum from play.

Instead of giving something to Troy:
        say "Troy sneers at you. 'I'm not interested,' he says coldly."
```

If you add this code to your game, you'll discover a couple of things. First, Troy will eat the plum, but will sneer at any other gift you may offer. (And incidentally, Inform knows that "offer" is a synonym for "give".) Second, if you try GIVE HAMMER TO TROY, the parser will cause you to pick up the hammer first before the giving action is attempted. This is called implicit taking. It's a handy feature, because it saves the player a bit of trouble: You don't need to PICK UP THE HAMMER first before giving it to Troy. (See the Action Processing Summary on page 146 to learn more about implicit actions.)

If the implicit taking fails (for instance, if you try to give an NPC a canary when the canary is in a locked cage), the giving action will never be attempted. Inform assumes that the PC can only give something to an NPC while actually holding it.

This makes sense for the giving action (though you'd need to write some extra code in order to allow the player to give an NPC a poisonous snake while the snake is curled up inside a cage). But the same condition applies to the showing action, and here it makes a bit less sense. What if I want to show an NPC the beautiful sunset? By default, Inform will try to have the player take the sunset first, and of course that won't work.

We can get around this limitation by writing a Before rule (instead of an Instead rule), like this:

```
Before showing the tuxedo to Troy:
        say "'Yes,' he replies sarcastically. 'I've already noticed that I'm wearing it.'" instead.
```

Now you can show the tuxedo to Troy without having the parser try to take the tuxedo (which of course Troy won't let you have).

Inform's default handling of ASK ABOUT and TELL ABOUT is good in one way, but bad in another way. These commands are implemented so as to handle topics, but they don't know about objects in the game. Let's suppose we want the player to be able to ask Troy about that interesting tuxedo. Here's the basic way to do it:

```
Instead of asking Troy about "tuxedo":
	say "'Oh, you've noticed my tuxedo,' he replies. 'I'm rather fond of it.'"
```

This is fine as far as it goes. The good news is, we can ask Troy about abstract topics, such as the meaning of life, using exactly the same syntax. The meaning of life doesn't have to be an actual object in our model world. The disadvantage is that the parser will only match the *exact* word or words in the topic. A quick look at the output will show why this can be a big problem:

```
>ask troy about tuxedo
"Oh, you've noticed my tuxedo," he replies. "I'm fond of it."

>ask troy about the tuxedo
There is no reply.
```

The Instead rule above didn't include "the tuxedo" as a text, and Inform has no idea that the topic "the tuxedo" is the same as the topic "tuxedo." Topics are not handled with the same kind of automatic intelligence as objects, because it wouldn't be practical for the parser to do so.

We can get around this limitation in a couple of ways. First, we can do it by letting Inform know about all of the ways we think the player might refer to the tuxedo. But it quickly gets a bit awkward:

```
Understand "tuxedo", "the tuxedo", "his tuxedo", "suit", "the suit", and "his suit" as "[tuxedo]".

Instead of asking Troy about "[tuxedo]":
	say "'Oh, you've noticed my tuxedo,' he replies. 'I'm rather fond of it.'"
```

Here we've created a new grammar token, "[tuxedo]", and provided an understand rule that spells out the words and phrases we want the player to be able to use. But of course, the careful author will already have given a vocabulary list to the tuxedo itself:

```
Troy is wearing a tuxedo. The description of the tuxedo is "Troy's tuxedo is a handsome bit of custom tailoring." Understand "Troy's", "handsome", "custom", "tailoring", "suit", and "attire" as the tuxedo.
```

Wouldn't it be nice if we could use that same vocabulary list while asking or telling Troy about the tuxedo? Inform lets us do this, and with only a bit more work. We need to create two new actions, quizzing it about and informing it about. These will use the same command words (ask and tell) as Inform's built-in actions, but we'll set them up to respond to "[something]", that is, an in-game object, instead of an abstract topic.

```
Quizzing it about is an action applying to two things. Understand "ask [someone] about [something]"
```

```
and "quiz [someone] about [something]" as quizzing it about.

Check quizzing it about:
	say "[The noun] shrugs unhelpfully."

Informing it about is an action applying to two things. Understand "tell [someone] about [something]"
and "inform [someone] about [something]" as informing it about.

Check informing it about:
	say "'That's interesting,' [the noun] says, stifling a yawn."

Instead of quizzing Troy about the tuxedo:
	say "'Oh, you've noticed my tuxedo,' he replies. 'I'm rather fond of it.'"
```

Notice that we've changed the Instead rule so that it now applies to the quizzing it about action, not to the built-in asking it about action. Now the player can ask and tell any character about any object in the model world, and use or not use THE in the input.

But there's still one big limitation we have to be aware of: The grammar token "[something]" will only match things that are visible in the room — that is, objects that are in scope. If we want to be able to ask and tell characters about objects that are not present, we need to change "[something]" in the grammar for our new actions to "[any thing]". We also need to change the definitions of our actions so that instead of applying to two things, they apply to one thing and one visible thing.

The reason for the latter change may not be obvious, but if you try editing your code to refer to "[any thing]" but don't change "thing" to "visible thing", you'll see that the action doesn't work properly. **Important concept:** Technically, "a thing" is shorthand for "a visible, touchable thing". That's what you'll be typing most often, so Inform lets you save a little typing. But when you say "visible thing," Inform understands that you're referring to something that's visible, whether or not it's touchable.

Here are the final versions of our new actions:

```
Quizzing it about is an action applying to one thing and one visible thing. Understand "ask [someone]
about [any thing]" and "quiz [someone] about [any thing]" as quizzing it about.

Informing it about is an action applying to one thing and one visible thing. Understand "tell [someone]
about [any thing]" and "inform [someone] about [any thing]" as informing it about.
```

Using this code, we can let the player ask characters about any object in the model world. We can also create, if we like, offstage things that can be used for conversation. These things don't need to be anywhere in the model world, and they don't need any properties other than their vocabulary:

```
Weather is a thing. Understand "clouds", "rain", and "wind" as weather.
```

But the idea of talking to an NPC about something that's not in scope raises a new complication: What if the NPC has never seen the object and thus knows nothing about it? For that matter, what if the player character has never seen the object and thus doesn't officially know that it exists? To deal with these questions, we need a way to track characters' knowledge.

Fortunately, there's already an Extension package that does some of the heavy lifting for us. Now that you've learned the basics of how to create an ask/tell/give/show conversation system, you can forget about the details unless you happen to need them for some reason. Download and install Conversation Responses by Eric Eve. This Extension provides a simpler way to write conversation topics. It also lets you write greetings and farewells.

I also like Eric's Conversational Defaults (which requires his Conversation Framework, so you'll need to download that too). Conversational Defaults makes it easier to set up default responses, which will be printed out in the game when you haven't written a response for a specific ask or tell action. With only a little effort, you can use Conversational Defaults to make your characters seem somewhat more lifelike than if the player only encountered the standard line, "There is no reply." With Conversational Defaults, we can give an NPC a bunch of possible outputs when he or she doesn't have a real response, and instruct Inform to select a response at random. Here's a short sample game that shows how to do this:

```
Include Conversational Defaults by Eric Eve.

The Test Lab is a room. The description is "Many devious tests are conducted here."

The zombie is a man in the Lab. "There's a dead guy sort of standing here. A zombie, from the look of
him." The description is "The zombie looks really ill. Really, really ill." Understand "dead", "guy",
"man", and "ill" as the zombie.

default ask-tell response for the zombie:
	say "[one of]The zombie shrugs minimally[or]The zombie scratches his cheek and grunts
unintelligibly[or]'I can't remember hardly anything,' the zombie says[or]The zombie only leers at you
menacingly for a moment before sliding back into his usual morose lethargy[at random]."
```

The effect we're creating above — having the zombie give a variety of different "non-responses" when asked or told about something for which the author hasn't written a real response — goes a long way toward making an NPC seem a bit more real.

Conversations, Part III: Character Knowledge

During the course of a game, an NPC might learn something, and what the NPC knows might change how he or she responds during a conversation. For instance, let's suppose the PC has witnessed Aunt Mary's house burning down, and is now in conversation (in a different location) with Uncle Jack. Uncle Jack won't know about the fire until the player tells him — so the command ASK JACK ABOUT MARY might need two different outputs, depending on whether or not Uncle Jack knows that Aunt Mary is now without a home.

The Extension called Epistemology by Eric Eve won't take care of this for us, because it's intended to track what the PC knows, not what an NPC knows. In this example, though, we're going to use Conversation Responses by Eric Eve. Conversation Responses includes Epistemology, which is convenient for this example, because the player can't ASK JACK ABOUT MARY unless the player has previously seen or is familiar with the Aunt Mary object. Both the “seen” and “familiar” properties are defined in Epistemology.

```
Include Conversation Responses by Eric Eve.

Aunt Mary is a woman. Aunt Mary is familiar.

The Living Room is a room.

Uncle Jack is a man in the Living Room.

Jack-knows-about-fire is a truth state that varies. Jack-knows-about-fire is false.

Response of Jack when told about "fire":
	if Jack-knows-about-fire is false:
		say "'I saw Aunt Mary's house burn down,' you tell Uncle Jack.[paragraph break]'Oh, no!' he cries. 'All those lovely antiques, burnt to a crisp!'";
		now Jack-knows-about-fire is true;
	otherwise:
		say "'I already told you about the fire, didn't I?' you say.[paragraph break]'Yes, yes,' Uncle Jack says, shaking his head sadly."

Response of Jack when asked-or-told about Mary:
	if Jack-knows-about-fire is true:
		say "'What do you suppose will happen to Aunt Mary now?' you ask.[paragraph break]'I guess she'll have to move in with her daughter,' Uncle Jack replies.";
	otherwise:
		say "'Isn't it too bad about Aunt Mary?' you comment.[paragraph break]'My goodness,' Jack exclaims. 'Did something happen?'[paragraph break]'Yes, her house burnt down,' you tell him.";
		now Jack-knows-about-fire is true.
```

In this example I’ve created Jack-knows-about-fire as a truth state that varies. In a game where the software needs to keep track of an NPC’s knowledge on a variety of subjects, keeping the data in a table (see p. 241) might be easier.

If you study this code for a minute, the way it works should become clear. The player can tell Jack about the fire (but not ask about the fire, since Jack only knows what the player has told him). The player can either ask or tell Jack about Mary. Both of these conversations test whether Jack-knows-about-fire is true. If it’s false, then a conversation will follow in which Jack learns about the fire.

Conversations, Part IV: Menu-Based Conversation

In a menu-based conversation, the player who starts talking with an NPC is presented with a menu of options, and selects one of the responses by number. The output might look something like this:

```
>talk to madelyn
You could:
[1] compliment Madelyn on her hair.
```

```
[2] complain that the gunfire in the street kept you awake all night.
[3] ask Madelyn why Uncle Jack was sent to prison.
[4] say goodbye.

>1
“You have really lovely hair,” you tell Madelyn.

“Do you like it?” she replies, patting a stray strand into place. “I’m not
sure avocado green is really a good color on me.”
```

By typing a “1” at the prompt, the player selects the first item in the conversation menu. What usually happens is that after the output of item 1 is printed, the menu comes back again, but now with item 1 removed. Eventually, there will be nothing left in the menu except “say goodbye.” The player will select that item, read the output, and the conversation is finished.

One advantage of this type of conversation system is that it can read in a more realistic way than bare commands like TELL MADELYN ABOUT HAIR and ASK MADELYN ABOUT UNCLE JACK. Another advantage is that the player is automatically directed toward topics of conversation that may be interesting or important to the story. But there are two problems. First, setting up this type of conversation system is not quite as easy as creating an ask/tell system. Second, the player will most likely just go through all of the possible topics one by one to see what they say. This process is called “the lawnmower effect,” because it turns the game-play into a rather boring mechanical process of mowing down everything on the menu.

If you look on **p. 7.8** of the Inform Recipe Book, “Saying Complicated Things,” you’ll find several Examples of conversational systems. Example 267, “Sweeney,” provides a hybrid system in which the player can ask or tell about topics, but will sometimes be prompted with a numbered list of items. The Recipe Book doesn’t give an example of a full menu-based conversation system, reporting that “they can be long-winded to set up, and therefore none are exemplified here, but several have been released as extensions for Inform.”

Looking on the Extensions download page, I found two: Quip-Based Conversation by Michael Martin and Simple Chat by Mark Tilford. The documentation for both is rather hard to follow, but many people have used them successfully in games. I had to make one small change in the code shown in the documentation for Quip-Based Conversation to get it to compile at all. (This problem was solved by Victor Gijsbers. He found that it was necessary to change “greeting” to “greetings” in a couple of places.) Here’s a short sample game:

```
Include Quip-Based Conversation by Michael Martin.

The Graveyard is a room. The description is "Old tombstones, many of them moss-covered and tilting, stand on all sides."

The zombie is a man in the Graveyard. "There's a dead guy sort of standing here. A zombie, from the look of him." The description is "The zombie looks really ill. Really, really ill." Understand "dead", "guy", "man", and "ill" as the zombie.

The greeting of the zombie is greetings. The litany of the zombie is the Table of Zombie Conversation.
```

```
Definition: the zombie is ready to talk if there is an enabled of 1 in the Table of Zombie Conversation.

Table of Quip Texts (continued)
quip	quiptext
greetings	"The zombie makes a sort of gurgling noise and lifts one corner of his lip in what might possibly be intended as a smile[if zombie is ready to talk]. Feeling a little more confident, you engage him in conversation[otherwise] ... but at the moment you can't think of anything else to say to him[end if]."
discuss weather		"'Does it look like rain to you?' you ask innocently.
'Hope not,' the zombie replies. 'Parts of me fall off when it rains.'"
death		"'Say ... are you dead? What's up with that?'[paragraph break]'It's not so bad,' the zombie muses. 'There's worse things, I guess. Can't think of one offhand, though.'"
silence		"You listen silently. Sure enough, the zombie isn't breathing."

Table of Zombie Conversation
prompt		response	enabled
"You could make an innocuous comment about the weather."	discuss weather	1
"You could ask him what it's like being dead."	death	1
"You could say nothing."	silence 1
```

In a game with a lot of characters, the Table of Quip Texts could get quite long. You would probably want to give the items in the quip column names like Bob-greeting and Sarah-greeting to make sure the code doesn't get mixed up. (Reading Bob's response when the player is talking to Sarah would be pretty weird.) While the quip table is global — that is, shared by the whole game — each NPC will have his or her own Table of Conversation. Bob would have a Table of Bob Conversation and Sarah a Table of Sarah Conversation.

Each time the player is in the presence of the zombie, typing a number that links to one of the remaining quips will trigger the output in the Table of Quip Texts. The Table of Zombie Conversation keeps track of what has been said by setting the number in the enabled column to 0 after a quip has been used. But you could reset this to 1 in your own code if you want the quip to become available again.

Switching entries in the enabled column to 1 or 0 in your own code has at least two uses. First, you can enable different quips for the character in different parts of the story. For instance, when the player discovers the diamond ring, you might switch on a couple of quips for Sarah to let the player ask Sarah questions about the ring. Second, you can steer the conversation as it's going on. Sarah might mention a topic herself: For instance, if you ask her why she's sad, she might say, "I lost my diamond ring somewhere." As part of this quiptext, you could trigger a routine that would enable the diamond ring topic(s).

If the player goes away and then comes back, the NPC will still be ready to engage in conversation, unless all of the quips have been used up. The player can simply type a number, the conversation will resume. This is not too realistic, but it works smoothly.

Giving Orders to Characters

The standard command that players can use to give orders to NPCs is in the form BOB, EAT THE PANCAKE. The NPC's name is what grammarians call a "noun of address," and is followed by a comma. If you want your game to respond to the alternate forms ASK BOB TO EAT THE PANCAKE and TELL BOB TO EAT THE PANCAKE, one easy way to do it is with an "after reading a command" rule. This type of rule intercepts the player's input before it reaches the parser. You can change the player's input in whatever way you like. This could be a sneaky way of creating an infuriatingly difficult puzzle, but here we'll use it in a more friendly way:

```
After reading a command:
    let T be indexed text;
    let T be the player's command;
    replace the regular expression "tell bob to" in T with "bob,";
    replace the regular expression "ask bob to" in T with "bob,";
    change the text of the player's command to T.
```

The main thing to be aware of here is that if your NPC can be referred to as "Bob", "Uncle Bob", "Uncle", or "man", the After Reading a Command rule needs to process *all* of those possible inputs separately, because the parser hasn't yet had a chance to figure out what object the player's command is referring to. The words "man" and "woman" will be especially sticky, because you might have several different NPCs in your game that can be referred to as "man" or "woman".

Inform won't let you rewrite the rule so that it reads "After reading a command in the presence of Bob". But you can do it this way:

```
After reading a command:
    let T be indexed text;
    let T be the player's command;
    if Bob is in the location:
        replace the regular expression "tell bob to" in T with "bob,";
        replace the regular expression "ask bob to" in T with "bob,";
        replace the regular expression "tell man to" in T with "man,";
        [...and so on for any other words that can refer to Bob...]
    change the text of the player's command to T.
```

Now the player can give orders to an NPC using a variety of normal phrases. But there's a better way, which was suggested by Emily Short. This uses Inform's ability to match regular expressions — frankly not a topic that belongs in a handbook for new authors. (If you're curious, have a look at **p. 19.6** in the Documentation, "Regular expression matching.") The code below will handle giving orders to all of the characters in your game:

```
After reading a command:
    let N be indexed text;
    let N be the player's command;
    replace the regular expression "\b(ask|tell|order) (.+?) to (.+)" in N with "\2, \3";
    change the text of the player's command to N.
```

But by default, NPCs will refuse to follow orders. If you want an NPC to obey the player, you need to write a **persuasion rule**, as explained on **p. 12.4** of the Documentation, “Persuasion.” A persuasion rule can be as specific or as broad as you like — you can have all NPCs obey orders, or only have one NPC obey one specific order. Here’s how to do it with two specific actions. We’re going to allow Uncle Jack to take the jewels or give them to the player:

```
Persuasion rule for asking Uncle Jack to try taking the jewels:
        persuasion succeeds.
Persuasion rule for asking Uncle Jack to try giving the jewels to the player:
        persuasion succeeds.

Instead of Uncle Jack giving the jewels to the player:
        now the player carries the jewels;
        say "Jack hands you the jewels.";
        rule succeeds.
```

This code assumes that the jewels are in scope. Perhaps Uncle Jack is carrying them. If you need an NPC to give the player something that has only been talked about, but that isn’t in the room at the moment, you have a slightly more complex problem. The easy way to deal with this is to have the NPC carry the object, but make it a concealed possession, as explained on **p. 3.24** of the Documentation (“Concealment”):

```
Troy is a man in the Billiard Room. Troy carries a banana.

Rule for deciding the concealed possessions of Troy: if the particular possession is the banana, yes;
otherwise no.

Persuasion rule for asking Troy to try giving the banana to the player:
        persuasion succeeds.

Instead of Troy giving the banana to the player:
        say "Troy gives you the banana.";
        now the player carries the banana;
        rule succeeds.
```

When the banana is concealed, the player won’t be able to X BANANA, but the banana will still be in scope, so TROY, GIVE ME THE BANANA will work as expected.

Giving Orders that Involve Going Elsewhere

A recent discussion on the newsgroup explored how to give an NPC an order that requires going somewhere else and doing something there. The basic difficulty in this case is, in effect, that once the NPC has gone somewhere else, he or she can’t “hear” the player character. A secondary difficulty, which will arise once we solve the first one, is that when the NPC is carrying out an action in a different location, the parser will report it to the player, even though the NPC is invisible. The solution to the latter requires the Extension called Scope Control, by Ron Newcomb. (Ron also contributed to the development of this example.)

Here's a simple game in which the player can order Jeeves the butler to go somewhere else and do something — specifically, go north, get the cheese, and then go south again.

```
Include Scope Control by Ron Newcomb.

The Dining Room is a room. The Kitchen is north of the Dining Room.
Jeeves is a man in the Dining Room. There is a cheese in the Kitchen.

A persuasion rule for asking someone to try doing something:
	persuasion succeeds.

The block giving rule is not listed in the check giving it to rules.

The subject is a person that varies. The subject is the player.

Before asking someone to try doing something, change the subject to the person asked.

Before reading a command: now the subject is the player.

After deciding the scope of the player while parsing for persuasion:
	place the subject in scope.

Unsuccessful attempt by someone doing something when the location of the actor is not the location:
	do nothing.

Before answering someone that something when the location of the noun is not the location, stop the action.

Test me with "Jeeves, n then get cheese then s then give me the cheese".
```

In a real game, we probably wouldn't want to use such a sweeping persuasion rule. The variable called "the subject" is changed to Jeeves when the player gives Jeeves an order. Then the subject is placed in scope. As a result, Jeeves will "hear" all of the orders even when he's in a different room. The two last rules (Unsuccessful attempt and Before answering someone) prevent Jeeves' actions from being reported if he's in a different room.

Moving Characters Around

People don't always stay in one place. In the real world, people do sometimes stay put for long periods of time — for instance, the librarian sitting behind her desk, who won't leave for hours. Other times, though, people move from one place to another. There are several ways to imitate that behavior in interactive fiction.

A classic effect in the early days of IF was to have an NPC wander at random around the map. This isn't very realistic, because people usually have reasons for going places, but at least it adds a little variety to the game. An example of an NPC who might be expected to behave this way would be a butler moving from room to room in a large mansion.

In the simple example below, we’ll create a square 3x3 grid of rooms and then create an NPC who wanders at random. If you can’t tell from the code what the map will look like, switch to the Index World tab after compiling the game.

```
A boring room is a kind of room. The description of a boring room is usually "Not much to see or do
here."

Room 1 is a boring room. Room 2 is a boring room. Room 3 is a boring room. Room 4 is a boring
room. Room 5 is a boring room. Room 6 is a boring room. Room 7 is a boring room. Room 8 is a
boring room. Room 9 is a boring room.

Room 4 is south of room 1. Room 7 is south of room 4.

Room 2 is east of room 1. Room 3 is east of Room 2.

Room 5 is east of Room 4 and south of Room 2. Room 6 is east of Room 5 and south of Room 3.
Room 8 is east of Room 7 and south of Room 5. Room 9 is east of Room 8 and south of Room 6.

Bob is a man in Room 3.

Every turn:
	let D be a random direction;
	try Bob going D.
```

As a result of the Every Turn rule, Bob will try going some direction or other in every turn, but if no room exists in the random direction, the “try” statement will fail. Bob will stay where he is. If the player is in a room when Bob arrives, the game will report the fact. Likewise, if Bob and the player are in the same room, the game will report when Bob succeeds in leaving. The output might look like this:

```
Room 5
You can see Bob here.

Bob goes south.

>z
Time passes.

>z
Time passes.

Bob arrives from the south.
```

This is okay as far as it goes, but the outputs (“Bob goes south” and “Bob arrives from the south”) are not very interesting. If you turn on rules reporting using the RULES command, you’ll learn that these outputs are coming from the “describe room gone into rule.” Unfortunately, you can’t find this rule by searching the Documentation. Nor could I find it by scouting around in the Index Rules area. If you’re curious, you can download the PDF called Appendix-A from the Inform website and search for this phrase in that file. You'll find the rule, which is about 30 lines long and covers any actor (including the player) going anywhere. The text outputs shown above come from *library messages*.

Replacing the entire rule would require quite a bit of tinkering to figure out what it does; I wouldn't want to try it, because I'd probably end up with bugs. An easier method (slightly easier, anyway) is to install and include the Extension called Custom Library Messages by David Fisher. Custom Library Messages is an unusually complex Extension, and we're not going to pause here to explore all of its features. All we really want to do is make Bob's arrival and departure a little more interesting. Here's how to do that:

```
Include Custom Library Messages by David Fisher.

Table of custom library messages (continued)
Message Id                                   Message Text
LibMsg <say npc arrives>                     "[guy arriving]"
LibMsg <say npc goes>                        "[guy departing]"

To say guy arriving:
        say "[one of]Here comes old [the actor] again[or][The actor] saunters up to you[or]The ominous creaking of floorboards announces the arrival of [the actor][at random]".

To say guy departing:
        say "[one of]At the sound of a thin scream in the distance, [the actor] dashes away[or]You suddenly notice that [the actor] has wandered off somewhere[or][The actor] evaporates in a thin puff of smoke[at random]".
```

This code doesn't tell the player which direction Bob arrives from or departs to; that would require more tinkering with the message. But now we have a way to add a little variety to the output. Note the use of "[the actor]" in the text blocks. If we have three or four characters wandering around, these rules will print all of their names correctly.

Here's a way to do basically the same thing, without including Custom Library Messages. This was suggested by Emily Short, who cautions that it won't handle some complex situations involving NPCs in vehicles:

```
Report someone (called the traveler) going:
        if the traveler is in the location:
                say "[one of]Here comes old [actor] again[or][The actor] saunters up to you[or]The ominous creaking of floorboards announces the arrival of [the actor][at random]." instead;
        otherwise:
                say "[one of]At the sound of a thin scream in the distance, [the actor] dashes away[or]You suddenly notice that [the actor] has wandered off somewhere[or][The actor] evaporates in a thin puff of smoke[at random]." instead.
```

Page 7.13 in the Inform Recipe Book provides several examples of different ways to move NPCs around. Possibly the most interesting of these is shown in Example 184, "Latris Theon." Inform includes a path-finding algorithm — a built-in procedure that will allow an NPC to calculate the best route from his or her current location to any other location. Applying this to the example given earlier is not difficult: Replace the last two blocks ("Bob is in Room 3" and the Every Turn rule) with the following code:

```
Bob is a man in Room 3. The destination of Bob is Room 3. Persuasion rule for asking Bob to try
going vaguely: persuasion succeeds.

Every turn when the destination of Bob is not the location of Bob:
        let the right direction be the best route from the location of Bob to the destination of Bob;
        try Bob going the right direction.

A person has a room called destination.

Understand "go to [any room]" as going vaguely.

Going vaguely is an action applying to one visible thing.

Carry out someone going vaguely:
   change the destination of the person asked to the noun.

Report someone going vaguely:
   say "[The person asked] looks amused, but accepts the commission to go to [the noun]."

Carry out going vaguely:
   say "You're too thoroughly lost."
```

We've done a few new things here. First, "A person has a room called destination." Since Bob is a person, he now has a destination. Initially we make his destination Room 3, so he's happy to stay where he is. Next, we include a persuasion rule, so that Bob will respond to an order to go vaguely. (The persuasion rule will cause BOB, GO TO ROOM 7 to work the way we'd like it to.) The rest of the new code is directly copied from "Latris Theon."

Notice also the difference between the two Carry Out rules. One applies to *someone* going vaguely (in other words, not the player character), while the other applies to the PC.

Example 259, "Odyssey," is on the same page of the Recipe Book. This shows how to move an NPC along a route that we've created in advance. The NPC's travel will be interrupted temporarily if the player chooses to interact with her. One refinement I'd suggest adding to this example, if you try it out, would be to define a scene called Athena Traveling. I would then change the Every Turn rule like this:

```
Every turn when Athena Traveling is happening:
        if Athena is active:
```

This will allow you to keep Athena (or whatever NPC you're using) in one place until some specific event occurs. For instance, you might do this:

```
Athena Traveling is a scene. Athena Traveling begins when the player carries the laurel wreath.
```

Now Athena will stay put (perhaps lurking in her temple) until the player picks up the laurel wreath. When that happens, she'll set out on her journey.

The Extension called Patrollers by Michael Callaghan provides some extra tools with which to write NPCs who move from place to place.

Characters Who Follow the Player

Writing a character who will find the player and then follow the player around like a faithful puppy is not difficult. Example 37, "Van Helsing" (again on **p. 7.13** of the *Recipe Book*) shows how to do it. We can adapt this code slightly to the example above by getting rid of the code that allows the player to order Bob around, replacing it with this:

```
Bob is a man in Room 3.

Every turn:
        if the location of Bob is not the location of the player:
                let the way be the best route from the location of Bob to the location of the player;
                try Bob going the way;
        otherwise:
                say "'Hey, I'm bored,' Bob says. 'Let's go for a ramble.'"
```

The same suggestion I gave for the "Odyssey" example (just above) applies here. You probably don't want Bob following the player from the very beginning of the game. To make Bob behave in a way that fits your story, you would need to define a Scene, and then say "Every turn when Too-Friendly-Bob is happening". Then write a sentence that defines when Too-Friendly-Bob begins.

The Extension called Simple Followers, by Emily Short, provides easy ways to create NPCs who will follow the player (or other NPCs) and will start or stop following on command. However, this Extension doesn't work the other way: It doesn't let the player follow characters who have left the room.

Characters the Player Can Follow

There are at least two ways (and maybe more) to set up a game where the player needs to follow another character. First, the NPC may be standing in the room, saying, "Come on, follow me!" or something of the sort. Second, the NPC may just have left the room, and the player may now be wanting to follow the NPC to see where he goes.

The second situation is trickier than the first. This is because of the way Inform handles *scope*. As explained on p. 38, the player can normally only refer to objects that are in the same room (and also visible — an object in a closed container is invisible, for instance, unless the container is transparent). If the NPC has left the room, he won't be in scope, so FOLLOW BOB will produce the unhelpful output "You can't see any such thing." Well, duh — that was why I wanted to follow him!

Example 287, "Actaeon" (found on **p. 7.13** of the *Recipe Book*), shows how to allow the PC to follow an NPC who has left the room. This example is easy to customize, but it's worth including some of the code here to point out a couple of features:

```
Bob is a man in Room 1.

Every turn:
	let current location be the location of Bob;
	let next location be a random room which is adjacent to the current location;
	if Bob is visible, say "Bob saunters off toward [the next location].";
	move Bob tidily to next location;
	if Bob is visible, say "Bob saunters in from [the current location]."

Following is an action applying to one visible thing. Understand "follow [any person]", "chase [any person]", and "pursue [any person]" as following.

A person has a room called last location.

Check following:
	if the noun is the player:
		say "You run around in circles briefly, but there doesn't seem to be much point in that, so you stop." instead;
	if the noun is visible, say "[The noun] is right here." instead;
	if the last location of the noun is not the location, say "It's not clear where [the noun] has gone." instead.

Carry out following:
	let the destination be the location of the noun;
	if the destination is not a room, say "[The noun] has gone where no one can follow." instead;
	let aim be the best route from the location to the destination;
	say "(heading [aim])[line break]";
	try going aim.

To move (pawn - a person) tidily to (target - a room):
	change the last location of the pawn to the holder of the pawn;
	move the pawn to the target.
```

If you look at the Every Turn rule here, or just try it out in a test game, you’ll find that Bob moves to a new room in every turn. In the examples earlier in this chapter, Bob would choose a random direction every turn — but if there was no room in that direction, he wouldn’t move. The line, “let next location be a random room which is adjacent to the current location;” causes Bob to choose a random room that actually exists, and move to it.

We have to move Bob in a tidy way, or “tidily” (an ugly word, but useful), so that he’ll keep track of where he was last.

If you comment out the last line in the Check Following rule, the player will be able to follow Bob even after several turns have elapsed. The result may turn out to be slightly unrealistic, however, because the PC won’t necessarily take the route that Bob took. Instead, the PC will choose the best route to wherever Bob happens to be *now.* If Bob has traveled around to the other side of the map — if he left heading south but is now in a room to the north — this code will cause the PC to go north, which is rather odd. That’s probably why the original code in this Example only allows the follow command to be used immediately after the NPC has left the room. Storing an NPC’s entire route, so as to allow the player to follow (perhaps by tracking footprints or dropped breadcrumbs) would be a lot more work, so we’ll leave it as a coding challenge for advanced Inform authors.

Stage Business

A character will seem more “alive” if she spontaneously does things on her own, or seems to. Sometimes the character will do something that changes the model world (see “Character Actions,” below). But sometimes we just want to print out a reminder to the player — a message that doesn’t actually do anything, but that makes the scene seem more interesting. I call these reminder messages “stage business.”

In this example, Janice is the maid. She will seem to do something without really doing anything:

```
The Living Room is a room. "This fussy old-fashioned living room is filled with heavy furniture."

Janice is a woman in the Living Room. "Janice is busy tidying up." The description is "Janice is wearing a maid's uniform. She's bustling about, doing all sorts of things."

Every turn when the player is in the Living Room:
	if a random chance of 1 in 3 succeeds:
		say "Janice [one of]flicks dust off of the grand piano[or]runs the vacuum cleaner[or]plumps up the cushions on the sofa[or]straightens a painting[or]polishes the mirror above the mantel[at random]."
```

The Every Turn rule will only run when the player is in the room with Janice. (If Janice were moving around the house, we’d need to change this rule a bit). About 1/3 of the time, a random message will be printed indicating that Janice is busy doing her job.

Character Actions

A good way to make a character seem more alive is for the character to react in some way when the player does something. This reaction could range from a simple comment to an explosive act.

Let’s suppose that old Alexander Button seems to be asleep in his chair. But he isn’t asleep, really. If you pick up the piece of paper on the table, he’ll snatch it away from you:

```
Alexander Button is a man in the Living Room. "Old Alexander [if Alexander carries the will]eyes you keenly[otherwise]seems to be asleep[end if]." The description is "He must be at least 90[if Alexander carries the will]. He's wide awake, and looking at you in a doubtful way[otherwise]. He seems to be asleep[end if]."

The end table is a scenery supporter in the Living Room.
The will is on the end table. The description is "A legal-looking piece of paper. You can't read what's on it, because it's upside down."

After taking the will:
	now Alexander carries the will;
	say "Alexander's eyes fly open. 'So ... snooping, eh?' He snatches the will away from you."
```

To get Alexander to react, we use an After rule. An After rule is a good idea, because it allows Inform to do its usual processing of the TAKE PAPER command. Only if the command succeeds — if it's possible for the player to take the will — will Alexander react.

The NPC action above was triggered by a taking action. Sometimes, we may want a character to react to something he or she sees. For instance, the character might demand that the player give him an object the player is carrying. Here's a short game that shows how this might work:

```
The Dusty Street is a room. "The unpaved street of Tombstone, Arizona, runs east and west here. The Red Dog Saloon is to the north."

The Red Dog Saloon is north of the Dusty Street. "The sour smell of spilled whiskey permeates this rough-hewn room, which is dim after the sunlit glare of the street."

Sheriff Earp is a man in the Red Dog Saloon. "Wyatt Earp is standing at the bar." Understand "Wyatt" as Sheriff Earp. The description is "Earp looks lean and mean. He's wearing a badge."

The player carries a six-gun. Understand "gun" and "revolver" as the six-gun.

Every turn:
	if the player carries the six-gun and Sheriff Earp can see the six-gun:
		say "[one of]'I see you're packin['] a weapon,' the sheriff says. 'Packin['] a weapon in town is illegal. Best you hand it over.'[or]'Well, are you gonna give me that six-gun, or ain't you?' Earp stares at you meaningfully.[stopping]".

Instead of going in the Red Dog Saloon:
	if the player carries the six-gun and Sheriff Earp can see the six-gun:
		say "'Best you not be walkin['] out of here with that firearm,' Earp says. He adjusts his own gunbelt slightly, as if he's thinking he may have to draw down on you.";
	otherwise:
		continue the action.

Instead of giving the six-gun to Sheriff Earp:
	now Sheriff Earp carries the six-gun;
	say "You hand over the gun. 'Thanks,' Earp says. 'You can go now. But don't be startin' any trouble, you hear?'"

Test me with "n / s / give gun to Earp / s".
```

In this example, the work is being done by the Every Turn rule and the Instead of going rule. The player can do anything in the saloon, but can't leave until he has given the gun to the sheriff.

Combat

I’ve never used the Extension called Armed by David Ratliff, but some of my students like it. It creates several kinds of weapons, allows you to set the maximum amount of damage that will be inflicted by a weapon, and computes the health of each character based on the amount of damage they’ve suffered. If you only want to add some weapons to your game, it might be easier to do it yourself, because Armed prints out the health of each character when the character is examined, and the health of the player character when the player takes inventory. If most of your game doesn’t involve combat, the printout of characters’ health can be a bit distracting.

If all you need to do is set up one encounter in which some combat takes place, here’s some code that will get you started. Note that we need to create an action called “attacking it with”. Inform’s standard library includes attacking, but not attacking it with.

```
The player carries a sword and a sofa cushion.

The Duke’s dead body is a thing.

Attacking it with is an action applying to two things and requiring light. Understand "attack [something] with [something]" and "hit [something] with [something]" as attacking it with.

Check attacking it with:
    if the second noun is not carried by the player:
        say "You're not holding [the second noun]." instead.

Report attacking it with:
    say "Your aggressive action has no visible result."

Instead of attacking the duke with the sword:
    if a random chance of 1 in 3 succeeds:
        say "You hack the duke to pieces with the sword!";
        remove the duke from play;
        now the duke’s dead body is in the location;
    otherwise:
        say "The duke parries your clumsy thrust expertly with his rapier."
```

We’re using randomness in the Instead rule above to allow the attack to succeed sometimes and fail other times. In a real game, you might want to include a small chance that the duke will kill the player. Since we may want to do any of three things in our Instead rule, we can’t use the “if a random chance … succeeds” syntax, since that only allows for a positive or negative result. Instead, we’ll do it by creating a temporary variable, X, and assigning it a random value:

```
Instead of attacking the duke with the sword:
    let X be a random number from 1 to 5;
    if X is 1:
        say "You hack the duke to pieces with the sword!";
        remove the duke from play;
        now the duke’s dead body is in the location;
    otherwise if X is 2:
        say "The point of the duke's rapier enters your rib cage.";
        end the game in death;
    otherwise:
        say "The duke parries your clumsy thrust expertly with his rapier."
```

Moods

Real people can get into a variety of moods — friendly, bored, hostile, romantic, frightened, twitchy, and so on. Mood Variations by Emily Short provides an easy way to switch characters from one mood to another in the course of conversation. It's meant to be used with a conversation package, and it's set up to make it easy to switch a character's mood in the course of conversation, using the syntax "[set hostile]" within a quote. This will always set the mood of the person the player is currently talking to.

But if you'd prefer to do without this Extension, you can set the character's mood by hand using a statement like "now Dave is hostile". This way, Dave can still react to something that Susan says. Here's a short game, which makes not a bit of sense, but may give you a quick idea of what's possible if you include Mood Variations. The "asked-or-told about" syntax is defined in Conversation Responses. Notice that the first thing we do is define a list of moods. This will include all of the moods that *any* character can get into in the game. We don't create a separate list of moods for each character.

```
Include Conversation Responses by Eric Eve.
Include Mood Variations by Emily Short.

The moods are friendly, neutral, bored, hostile, and frightened. A person is usually neutral.

The Test Lab is a room. "Many devious tests are conducted here."

The player carries an apple and a pear.

Susan is a woman in the Lab. Dave is a man in the Lab.

Response of Susan when asked-or-told about the pear:
    if Susan is neutral:
        say "[set friendly]'That's quite a pear, there,' Susan replies.";
        now Dave is hostile;
    otherwise if Susan is friendly:
        say "[set hostile]'I love that you keep asking me that,' she says.";
    otherwise if Susan is hostile:
        say "'Do shut up,' Susan snaps."

Response of Dave when asked-or-told about the apple:
    if Dave is neutral:
        say "[set bored]'I saw one once, in Eden,' Dave replies.";
    otherwise if Dave is bored:
        say "[set hostile]Dave yawns ostentatiously.";
    otherwise if Dave is hostile:
        say "'Are you trying to pick a fight?' Dave asks.";
        otherwise:
                say "'Apples, apples....'"
```

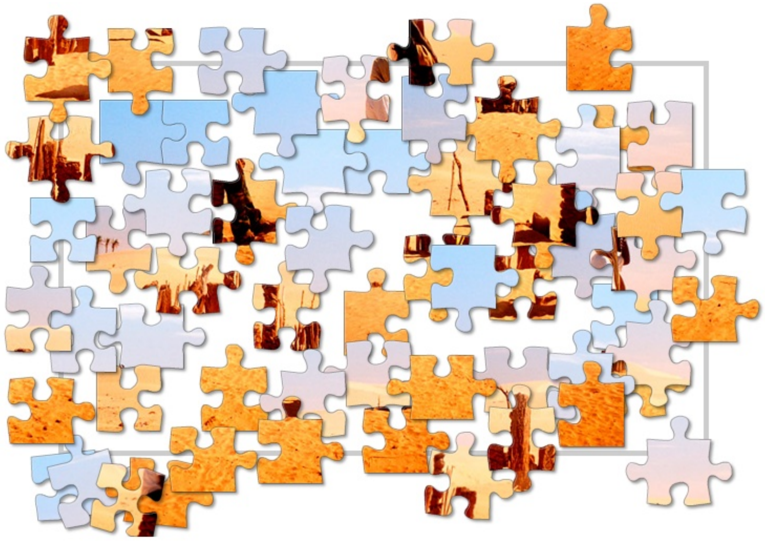
Body Parts

Giving your characters body parts is not always necessary, but if the description mentions something prominently (Janet's blue eyes, the duke's hawklike nose, or whatever), you can easily use Inform's ability to create parts of objects (as discussed in Chapter 3 of this *Handbook*) to make the body parts. In some games, you might want to give all of your characters body parts. In this case, you would make the part a kind of thing, like this:

```
The Living Room is a room.

A nose is a kind of thing. A nose is part of every person. The description of a nose is usually "Seen
one nose, seen [']em all."

Susan is a person in the Living Room.
The description of Susan's nose is "It has a cute upward tilt." Understand "cute" as Susan's nose.

The description of your nose is "You can't see much of it, as you're behind it."
```

The only odd thing about this is that you can't say "The description of the player's nose". Or rather, you can; it will compile; but it won't work. Instead, you have to say "your nose". Inform will understand that the player can then refer to this object as "my nose".

Chapter 6: Puzzles



In the short section on puzzles in Chapter 1, I pointed out that a few games are "puzzleless," but that most games have a few puzzles, or a lot of them. The point of including puzzles is that the player becomes a sort of collaborator in the unfolding of the story. The story will only move forward when the player takes some action.

Almost any sort of action might be required — watering a tiny plant, for instance, might turn it into a giant beanstalk that can be climbed. But possibly the most basic form of puzzle in IF is the locked door. The player doesn't yet have the key that unlocks the door, and has to hunt for the key. Unlocking and opening the door will give the player access to a new room, or perhaps to an entirely new region of the model world, containing dozens of rooms.

The way I look at it, every puzzle in every game, no matter what form it might take, is ultimately not very different from a locked door. You can't see what's on the other side until you find the key, whatever the key is. The "key" might be an apple that you need to give to the wicked witch so she'll go to sleep so you can rescue the children in the oven — in some sense, the apple is a key. But that may not matter much: the code you'll write to handle the command GIVE APPLE TO WITCH will probably look somewhat different from the code for UNLOCK DOOR WITH KEY.

In this chapter we'll take a look at many of the common types of puzzles, and suggest a few ways to implement them. This is not, I hasten to add, a complete inventory of puzzle types. It's intended merely to introduce a few ideas that new IF writers may not have considered.

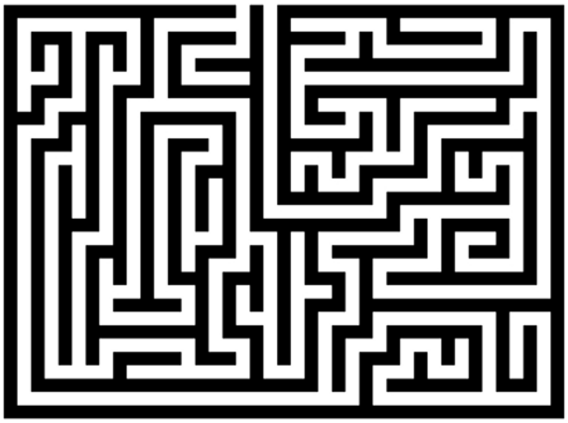
IF authors have written lots of essays and tips about puzzles. Stephen Granade has a good short essay, for instance, on his Brass Lantern website (http://brasslantern.org/writers/iftheory/betterpuzzles-a.html). Graham Nelson's manual on programming Inform 6, the *Designer Manual 4* (usually called the *DM4*), has a very good section on designing puzzles; currently this is to be found at http://www.inform-fiction.org/manual/html/s50.htm.

I like to include some easy puzzles in a game, and a few hard ones. But as a player, I prefer the easy ones, because I'm very bad at puzzle-solving. While playing other people's games, I often get stuck. Many authors feel that it's a good idea to help players by including some form of hint system in the game. You'll find some advice on how to set up a hint system in Chapter 10. Other authors feel that providing hints spoils the game, or is just too much trouble.

Mapping

Creating and maintaining a map that shows the layout of the game isn't much of a puzzle, but it definitely qualifies. The player who neglects to create a pencilled map on which she notes that there's a south exit from the Library is quite likely to miss something important in the game.

Many games today include a full list of available exits (or at least, of the obvious exits — see below) as part of the room description. In some older games, this convention wasn't adhered to: exits might not be mentioned at all. Trying all of the available compass directions, only to be told over and over, "You can't go that way," is not much fun, which is why this type of puzzle isn't much used anymore.



The classic example of a mapping puzzle is a maze. A maze is a set of rooms (usually between ten and fifty of them) in which the player can move freely, but in which navigation is difficult for some reason. Most players don't like mazes, because mazes can be mind-numbingly repetitive to solve.

The original game of Adventure had two mazes. The room descriptions within each maze were all identical (and no exits were listed), so just LOOKing wouldn't tell you where you were. The connections between rooms in a maze is likely to be "twisty," which means that if you go north from room A and arrive in room B, going south from room B quite likely won't take you back to room A. Some room connections may even fold back on themselves: When you go east from room A, you may end up back in room A! This type of navigation system is easy to set up in Inform (see "Twisty Connections" in Chapter 2 of the *Handbook*).

The time-honored way to map this type of maze is to drop an object in each room as you go along. The pattern of objects differentiates the rooms from one another. If you put this type of maze in a modern game, it's safe to say you'll amuse nobody. But there are many ways to make mazes interesting (at least, they'll be interesting to people who don't mind mazes). In "A Flustered Duck," I included a maze that could only be mapped while the player character is wearing a blindfold. When the blindfold is not worn, the connections between rooms are entirely random and the room descriptions are all identical — but when the PC is wearing a blindfold, he can find distinct room connections using his sense of touch.

Blocked Passageways

A locked door is obviously a type of blocked passageway. In "The Craft of Adventure," Graham Nelson observes, "Almost invariably games close off sections of the map (temporarily) by putting them behind locked doors, which the player can see and gnash her teeth over, but cannot yet open. And almost every variation on this theme has been tried: coded messages on the door, illusory defenses, gate-keepers, the key being in the lock on the wrong side, and so on. Still, the usual thing is simply to find a key in some fairly remote place, bring it to the door, and open it."

Quite often you'll find several locked doors in a game, and several keys. This can be at least mildly amusing, because the player usually has the wrong key. Finding a good place to hide the key isn't always easy. Experienced players will try LOOK BEHIND and LOOK UNDER with any object (such as a bed, painting, or rug) that looks as if it might conceal a small object. You can also expect players to SEARCH anything (such as a couch with cushions) that looks to be a likely place where something might be hidden. For more on how to set this up, see Chapter 3, "Things."

Popular variants on the locked door include chasms too wide to leap across (with or without a bridge too flimsy to support your weight), doors guarded by uncooperative characters who won't let you pass (see "The Uncooperative Character," below), and secret doors. The secret door appears not to be a door at all (see "Hidden Items," below), so the "unlocking" process is mainly a matter of finding the door.

The inventory-blocking passage is a related type. In this puzzle, you can traverse a passageway freely — but not while carrying certain items. And of course, you'll almost certainly need those items once you get to the far side of the passageway, so this can also be considered an inventory manipulation puzzle. A fiendish inventory-blocking passageway, which (unless my memory is playing tricks on me) dates clear back to "Adventure," is a passage through which a powerful wind is blowing — a wind that will blow out your lamp or candle. This puzzle is a cross between the inventory-blocking passageway and the darkness puzzle (see below).

It's worth noting, however, that not all cases in which inventory can't be carried through a passage are puzzles. In some games, the author chooses to manage the player's inventory by making each area of the map self-contained. Inventory items may be dropped automatically when you use the passageway, or the passageway may remain blocked until you get rid of what you're carrying. (It may not always be obvious, though perhaps it ought to be, whether the blockage is a puzzle or just the author's inventory management system at work.)

The ultimate blocked-passageway puzzle is a single room from which there appears to be no exit at all (except perhaps a locked door to which you lack the key). This room is often a prison cell, but other variations are common. The solution may involve inspecting all of the items in the room *very* carefully or conversing with someone who is on the other side of the wall.

Darkness

The need to bring a light source into a dark room or rooms is a puzzle that dates clear back to "Adventure," which of course was set in a system of caves. See the section on "Dark Rooms" in Chapter 2, page 68. To solve a dark room puzzle, the player has to find and carry an object that provides light. Today, most rooms in most games are lighted by default, but dark rooms still show up in quite a few games. The light source that's needed may be unusual, or may last for only a limited number of turns — a burning match, for instance — making the dark room a timed puzzle. (See

“Timed Puzzles,” below.)

A variant of the dark room puzzle is a dim room, in which you can see some things, probably large ones, but not small things. You might be unable to read written materials in the dim room, for instance.

Objects dropped in dark rooms can’t be picked up again. This fact requires careful thought; it’s possible to make your game unwinnable without meaning to — for instance if the player drops the matches in the dark room before finding the candle. If you don’t want to allow the game to become unwinnable, you’ll need to write a rule that will prevent this from happening:

```
Instead of dropping the candle:
	if the player is in a dark room:
		say "Better keep your hands on the candle for now. You may need it later.";
	otherwise:
		continue the action.
```

Hidden Items

Experienced players of IF know to EXAMINE everything that’s mentioned in the room description, or in a description of another object. So it’s fair to tuck clues to various puzzles in descriptions. Quite often, the room will contain an object which is itself scenery, but which will reveal further details when examined. The description of a wall panel, for instance, might suggest to the player that the panel is a secret door. Or maybe the panel looks entirely innocent when examined, but if the player examines the rug he’ll learn that a semicircular mark on the rug curves out from the wall panel. (If you’re going to do that, be sure to mention the rug in the room description.) The details are “hidden” only in a technical sense, because you have to examine something else in order to notice them, so this is only barely a puzzle. Many games that award points don’t award any points for examining things, because the action is just too easy and obvious.

Examining won’t always reveal hidden items, however. As a player, you’ll want to get in the habit of looking under and behind anything large. Containers may need to be searched in order to reveal what’s hidden in them. By default, Inform will list the contents of any open container when the player examines the container, but as an author you might want to make the puzzle a tiny bit more difficult. If you do this, though, it would be a courtesy to the player to provide a clue that more action needs to be taken. Here’s an example:

```
The big old box is an open container in the living room. The big old box is scenery.

The sponge is in the box.

Instead of examining the big old box:
	say "It's just a big old box. It's so large, in fact, that a casual glance doesn't tell you much what might be lurking down there in its depths."
```

The result, when the game is played, looks like this:

```
>x box
It's just a big old box. It's so large, in fact, that a casual glance
doesn't tell you much what might be lurking down there in its depths.

>look in box
In the big old box is a sponge.
```

The LOOK IN command causes the same action as SEARCH.

A sorting puzzle probably qualifies as a subtype of the hidden item puzzle. In a sorting puzzle, there are a great many similar or seemingly identical objects, all available at the same time. Your goal is to find a way to distinguish the one you want from all of the others. For an especially fiendish example of this, you might want to download and play the game “69,105 Keys.” It’s a one-room game with a locked vault, and, you guessed it, 69,105 keys, only one of which opens the vault.

Items Out of Reach

Putting an important object on a high shelf or at the bottom of a narrow hole, where the object can be seen but not touched, is a common puzzle. The player may need to stand on a chair to reach the shelf, or find a giant magnet on a rope to retrieve the iron key from the hold. Sometimes the solution is to go around to an entirely different location, from where the shelf or hole bottom is reachable. Sometimes the solution is to shake the shelf so that what’s on it will fall off.

Locked Containers

As with a locked door, the usual way to open a locked container is to find the key. Sometimes the locked container has a glass door, so that what’s in it can be seen but not reached. (The way to do this in Inform is to make the container *transparent*. This word has a special meaning to the compiler and the parser.) Once in a while the solution is simply to break the container.

Combination Locks

Either a door or a locked container may have a combination lock instead of a physical key. The most straightforward type of combination lock puzzle is, perhaps literally, a combination lock. It may have a single dial that needs to be turned to three or four numbers in succession, or a row of dials each of which needs to be set to a specific number. The number of possible combinations is likely to be large, so the only practical way to solve the puzzle is to find the piece of paper on which some thoughtful or absent-minded person has scribbled down the combination.

A more interesting combination lock puzzle can be created by scattering clues around. For instance, the dials might be of various colors, and each dial might have a number of symbols on it. Somewhere in the model world you might find the color red associated with a bird, which would be a clue that the red dial should be turned to the bird symbol. (“Blue Lacuna,” by Aaron Reed, has a puzzle of this type.)

Large mechanisms such as revolving rooms can operate as combination locks: To use one or more of the exits from the room, you'll need to figure out how the buttons and levers change the orientation of the room. In another variant, the buttons and levers might not be in the same room as the lock mechanism itself: To solve the puzzle, you may need to travel from room to room to see what happened when you moved the lever.

In a more general sense, any puzzle that requires the player to perform two or more actions in a specific order qualifies as a combination lock. If the game includes a muzzle-loading rifle, for instance, the player will need to pour the powder into the barrel, place a patch on the end of the barrel, put the ball in the patch, and then use the ramrod to push the ball and patch down into the barrel. If the player puts the bullet into the barrel first and then the powder, the rifle won't fire. (The real procedure also requires measuring the powder, tapping the stock against the ground to settle the powder, and using a bullet starter before the ramrod to get the ball started down the barrel. But that may be too much detail for a game.)

Manipulation Difficulties

Many games include objects that are too heavy to lift, too hot, too slippery, or difficult to handle for some other reason. This type of puzzle offers many opportunities for the author. Providing a handy oven mitt for picking up the hot object might be too easy: you might want to give the player a baseball glove instead.

Figuring out how to operate machinery can be easy, or almost endlessly complicated. In "The Craft of Adventure," Graham Nelson comments, "In some ways the easiest puzzles to write sensibly are machines, which need to be manipulated: levers to pull, switches to press, cogs to turn, ropes to pull.... They often require tools, which brings in objects. They can transform things in a semi-magical way (coal to diamonds being the cliché) and can plausibly do almost anything if sufficiently mysterious and strange: time travel, for instance."

Among the things one may find in a game that pose manipulation difficulties — and also implementation difficulties for the author — are fire, water, rope, and chemicals of all kinds. Chapter 10 of the *Inform Recipe Book* has some good examples of how to create puzzles of this type. You might want the player to mix three chemicals to create a new substance, for instance. Mixing would naturally require a container that the ingredients can be put into. (The cocoa mug puzzle in "Lydia's Heart" was easily the most complex and difficult puzzle to write in the game, and I'm still not sure it's free of bugs. The puzzle had three ingredients — a paper envelope full of powdered cocoa, a sleeping pill, and some water. The water could be either hot or cold, and once the pill or powder was combined with the water, it couldn't be extracted again. This kind of thing can be fun if it's well done, but it's far too easy to do it badly.)

Here's a simpler example of a manipulation difficulty puzzle, which was suggested by a post from Susie on the newsgroup. The player can only see things while wearing glasses. To implement this, we need to intercept both the LOOK and EXAMINE actions. It also seems advisable to stop the player from picking things up while not wearing the glasses. But if we do that, we also have to be careful to allow her to pick up the glasses while not wearing them — otherwise, dropping the glasses will make the game unwinnable.

```
The player is wearing some glasses. The description is "They're as thick as bottle-bottoms."
Understand "bifocals" and "spectacles" as the glasses.

Instead of looking when the player is not wearing the glasses:
        say "Unfortunately, you're as blind as a bat without your glasses."
Instead of examining something when the player is not wearing the glasses:
        try looking instead.

Instead of taking something when the player is not wearing the glasses:
        say "You grope around, but you can't find a darn thing."

Instead of taking the glasses when the player is not wearing the glasses:
        if the player does not enclose the glasses:
                say "You grope around blindly and somehow find your glasses.";
                now the player carries the glasses;
        otherwise:
                say "You've already got them."
```

Because the instead rule about taking the glasses is more specific than the instead rule about taking something, it will be listed earlier in the instead rulebook. This is what we want: The player will be able to pick up the glasses.

A subtype of manipulation difficulties is what we might call the strapped-to-a-table puzzle. I've seen this puzzle presented at the beginning of a game, but it might happen at any point, especially after the player character has been knocked unconscious. The puzzle is, you don't seem to be able to do *anything.* The repertoire of actions normally allowed in IF is all (or seemingly all) disabled. In Emily Short's one-room game "Glass," for instance, you're a bird on a perch. Your wings have been clipped, so you can't even fly. Eventually the player discovers that there's one thing that the bird *can* do. But I'll let you discover for yourself what it is.

Enigmas

A commentator on rec.arts.int-fiction (I've forgotten who) made an interesting distinction between puzzles and problems. Problems, he maintained, are difficulties that are intrinsic or sensible in the world of the story. By that definition, most of what we've been looking at in this chapter are problems, not puzzles. A puzzle-type puzzle, in contrast, is something that doesn't have any business being in the world of the story: It's there strictly to give the player some mental gymnastics. If we're going to keep using the word "puzzle" in a broader sense, to refer to all of the problems that the PC may face during the story, we might call this type of puzzle an enigma.

The ultimate example of an enigma is the game “Gostak” by Carl Muckenhaupt. In this game, *all* of the nouns and verbs, including the ones you use in commands, have been replaced by nonsense words of the author’s own devising. The puzzle is to figure out how to read the text of the game and issue commands. Roger Firth’s “Letters from Home” is also an enigma-based game, with only a thin veneer of story.

Deceptive Appearances & Unusual Usages

An object with a deceptive appearance is different from an object whose description is poorly written (see “Inadequate Implementation,” below). The initial description of an object might be vague because the PC has never seen anything like it and doesn’t know what it is, or because only part of it is present, thus giving it an enigmatic shape, or because it is in an odd condition.

As a simple example, consider an object called “a wadded-up piece of paper”. This is quite likely a deceptive appearance. If you try READ PAPER, you’ll probably be told, “You can’t read a wadded-up piece of paper!” However, commands like SMOOTH PAPER and UNFOLD PAPER will probably change the name of the item to “a slightly wrinkled note”, after which READ PAPER or READ NOTE will reveal what’s written on it.

A variant of the unusual usage puzzle is what we might call the unexpected action puzzle. This isn’t quite the same as a guess-the-verb puzzle, because the author has done it on purpose. Suppose, for instance, that you’ve got a little glass bottle with some tablets in it. The top of the bottle is stuck tight, so OPEN BOTTLE simply doesn’t work. However, BREAK BOTTLE might be the intended solution of the puzzle. This particular example also fits in the locked container category. A better example might be an object described as “a small tarnished piece of metal.” The command POLISH METAL might turn this object into a mirror which which you could direct a beam of light. (To implement this, you’d also want to change the object’s printed name property.)

Assembly Required

Sometimes objects have been taken apart, and the player has to put them back together in order to use them. In Chapter 3 of this *Handbook* we looked at a postcard and a stamp. The stamp could be attached to the postcard. If this is a puzzle, maybe the player would then have to put the postcard into a slot in the Post Office. If the stamp hasn’t been attached, the message will never be delivered.

Another simple example would be a length of metal pipe that can be inserted into an appropriately positioned hole in a machine. When the player does this, the piece of metal becomes a crank that can be turned.

The relationships among the parts may be obvious — a lightbulb that needs to be put in a lamp, for instance. Or the player may be called on to improvise by assembling objects that have no obvious relationship to one another.

"The King of Shreds & Patches" has a very nice assembly puzzle in the form of a flintlock pistol that has to be loaded before it can be fired. The materials (powder, lead balls, and so on) are all readily at hand, but the player has to figure out how to manipulate them. What's especially nice about this puzzle is that once the player has done it the first time the hard way, the command LOAD PISTOL can be used as a shortcut to do it again.

Mechanisms

Mechanical devices that have to be switched on, or that need power to operate, are common. A mechanical device may have several buttons or levers, and it may not be obvious what the buttons and levers do. They might seem to do nothing, or do nothing in the room that the player is in while having an important effect in a different room. Or the result of pressing a button or pulling a lever might be delayed for a few turns.

A mechanical device may have a gauge or data readout whose meaning is not clear. If you pull the red lever, for instance, the green dial might change to read "3". In this case, you may have to figure out what several levers do, and what the readings on the dials mean.

Vehicles

A vehicle in which the player can travel around will seldom be a puzzle in its own right. More often, it's an example of one or two other types of puzzles ― mapping and manipulation difficulties. That is, you'll need to figure out how to use the vehicle, which may involved finding the lost steering wheel or just understanding what the red and green buttons do; and then learn, probably by pressing the red button, where it can take you. On occasion you may have to fuel the vehicle, which would come under the heading "Assembly Required." You may find that some objects can't be carried in the vehicle, which would be an inventory-blocked passageway, or that the vehicle is required in order to carry certain objects from place to place — again, an inventory-blocked passageway. You may need to get out of the moving vehicle at a specific moment, which would be a timed puzzle.

Conveyances such as wheelbarrows, that have to be pushed from place to place, are used, again, mainly as a way of handling inventory objects.

Consulting Information Sources

In many games, key pieces of information are hidden in a large book, or possibly in a computer terminal of some sort. To get at the information, the player uses a syntax like LOOK UP KING HENRY IN HISTORY BOOK or LOOK UP POISONS IN COMPUTER. The solution to this sort of puzzle is to keep trying topics until you find an input that the information source responds to. Or the player may need to find another piece of information somewhere that says, in essence, "King Henry is listed on page 447 of the history book." Once the player is in possession of this information, TURN TO PAGE 447 should reveal the information.

Alternatively, the information may simply be written on a piece of paper in the game, or scrawled on a blackboard, and the puzzle is to find it. In some games, the information on the paper or blackboard is incomplete or difficult to understand. It may be written in code, or the paper on which it's written may have been torn up, forcing the player to collect the scraps in order to decipher the complete message.

Not all of the information sources in games are in the form of encyclopedias, computers, and carelessly dropped notes. If you see any murals, mosaics, paintings, or tapestries in a game, pay close attention to the images depicted on them. Likewise carvings. Sound and video playback devices are a bit less often used, but they do show up from time to time. The player may need to TURN ON TAPE PLAYER, for instance, followed by an esoteric command like FAST-FORWARD TO 447.

With an informal type of information source, it's usual that you won't know immediately whether you're getting useful information or just background on the story. This is especially true of NPCs, who may ramble on about all sorts of topics, their conversation containing disguised or ambiguous information.

The Uncooperative Character

People and animals can function as puzzles in several different ways. Guards and guard dogs, for instance, can be relied on not to let the player pass through a tempting doorway. The door isn't locked, but it might as well be. The player has to figure out a way to distract or make friends with the guard. Almost any character can act as a guard — a fussy librarian, for instance, who won't let the player into the computer room until the player shows her a library card.

Instead, a character might have an object that the player needs, and the puzzle might be how to get him or her to hand it over. Some characters can do things the player character can't, in which case the puzzle is how to get the character to follow instructions and perform some action. The player might be unable to retrieve an object at the bottom of a pool, for instance, not knowing how to swim — but a friendly dolphin might be willing to obey orders. (How a dolphin would pick something up … I'll leave you to work that out for yourself.)

Timed Puzzles

Generally speaking, time passes in IF only when you issue a command. (A few games may include real-time events, but this is not common.) In a timed puzzle, something is happening, and you need to interact with it within a specified number of turns. You may need to figure out the proper command, or even issue a whole sequence of commands, within a fixed time-frame.

The simplest type of timed puzzle is, of course, a lighted fuse attached to a bomb. Also simple (and dating back to the very first IF) is the light source that will expire after a fixed number of turns, perhaps because it's a battery-powered flashlight. The player who is left in a dark room when the battery dies is likely to be in serious trouble. In the early days of IF, games sometimes required that the PC eat and/or sleep on a regular schedule; failure to do so would have bad consequences. Eating and sleeping are no longer common in modern games.

The rowboat puzzle in "King of Shreds & Patches" is a complex timed puzzle. You're in a boat on a river, and the boat is being carried downstream by the current. Unless you visualize clearly how the boat is moving and issue the correct navigation commands, you'll be unable to reach the dock before the small hole in the bottom of the boat causes the boat to sink.

If you fail to take the proper action in a timed puzzle, the game will almost certainly become unwinnable, and quite likely will end in a sudden and spectacular way. Timed puzzles are, by their nature, somewhat more cruel than other types of puzzles, because the possibility of losing the game is more immediate.

A relative of the timed puzzle is what might be called a course-of-events game. In a game of this sort, you may be able to get from the start of the story to the end by typing 'z' over and over — no action is required, as the course of events will unfold without your taking any action at all. However, if you do that, you may well miss the point of the story entirely, or fail to reach the most desirable ending. Your opportunity to influence the course of events (and thereby steer the course of the story) will have passed unnoticed. The puzzle lies in figuring out exactly how your actions may influence the course of events. Asking certain questions of a character at certain times, for instance, may have a large effect — but it may not be obvious what to ask, or when. Emily Short's short game "Glass" provides an excellent example of this type of puzzle.

Inadequate Implementation

This isn't really a type of puzzle, except in the technical sense: It's a design flaw — and a serious one. In an inadequate implementation puzzle (which can be of almost any type), the player lacks necessary information because the author has neglected to include it, either in the output text or in the software code. The "puzzle" boils down to making random guesses or reading the author's mind, which is pretty much the same thing.

The most common subtype of the inadequate implementation puzzle is called a guess-the-verb puzzle.

You (the player) can see exactly what you need to do to solve the puzzle, but the author has failed to write code that handles any of the obvious and appropriate commands that you try. Very few things are more infuriating for an IF player than trying ‘stab guard with sword’, ‘cut guard with sword’, ‘kill guard’, ‘attack guard with sword’, and so on, being met in each case with a blank refusal by the game, only to find that the correct syntax is ‘swing sword’ while you’re in the room with the guard.

Another subtype is the inadequately described room or object. Describing the manner in which a complex mechanical puzzle is constructed is not easy, and it’s a place where many authors fall down on the job.

In his excellent essay “The Craft of Adventure,” Graham Nelson points out a related pitfall — the in-joke puzzle. You may know what scatological phrase is suggested by using the Greek letters in the name of a certain fraternity as an acronym, or that Petrarch wrote sonnets about a woman named Laura, or the month for which the birthstone is topaz (it’s November), but it’s unlikely your players will make the connection.

In writing your game, it’s vital that you think carefully about the design of your puzzles.

First, have you given the player enough information to enable her to solve the puzzle? Remember: Something that’s obvious to you (the author) may be mystifying to anyone who can’t read your mind.

Second, have you considered and built into your game all of the commands you can think of that a player might use while working on a puzzle? Almost as bad as guess-the-verb puzzles are misleading responses from the parser. Here’s an example:

```
>hit guard
Violence isn’t the answer to this one.

>hit guard with stick
You smack the guard in the head with the stick, and he goes down like a
sack of potatoes. Congratulations! Now you can steal the jewels from the
vault!
```

Inform’s default response to HIT GUARD is just plain wrong in this case, because violence *is* the answer to this one.

Chapter 7: Winning & Losing

Inform has no built-in concept of "winning" or "losing" — it's up to you to define how your game will end. The easiest ways to do this are with the phrases "end the game in victory" and "end the game in death". Both of these stop the game-play immediately. The first phrase produces the output

***** You have won *****

and the second the output

***** You have died *****

After the game has ended, the player will have the option to quit or restore a saved game. To see how "end the game" phrases work, try this simple example game:

```
The Forest is a room. "Tall trees stand on all sides."

The gold crown is in the Forest. The description is "The crown is studded with sparkling jewels!"

After taking the crown:
	end the game in victory.

The player carries a cyanide pill. The cyanide pill is edible.

After eating the cyanide pill:
	end the game in death.
```

An After rule is a good place to end the game, because it gives Inform a chance to make sure the action actually took place before the game ends.

It's usually a good idea to write a description of what the player has done to end the game before officially ending it. We might revise our code like this:

```
After taking the crown:
	say "At last! You've found the gold crown you've been seeking!";
```

```
        end the game in victory.

After eating the cyanide pill:
        say "Moments after ingesting the pill, you begin to feel very, very ill.";
        end the game in death.
```

This is fine as far as it goes, but sometimes the end of a game calls for something less drastic than the death of the player character. Depending on the story, maybe the game should detect that the player is giving up too soon. In that case, you could do this:

```
Instead of going north in the Forest:
        say "As you wander back down the mountainside toward town, you feel a keen and lingering
sense of regret.";
        end the game saying "You have missed the point entirely!"
```

After the phrase “end the game saying”, you can give whatever message you like. This will appear, in bold and surrounded by the rows of asterisks, when the game ends.

Oddly enough, “end the game” is a phrase that *must* include the word “the”. More often than not, Inform strips out “the” when compiling, but here it’s required. The phrase “end game in death” won’t compile.

A Word About Fairness

In the early days of interactive fiction, numerous games were released that would kill the player without warning. That was part of the fun (?) — when you opened a door, maybe an ogre would jump out at you and hit you with his club, and the game was over. If you had been smart enough to save your game position not too long before, you could restore the saved game and try to be more careful about the ogre the second time. If you hadn’t been smart, you might have to spend hours getting back to that game position again.

Today, many players and authors feel that this type of story event is more annoying than fun. The trend is to give the player some sort of advance warning, which might be subtle or obvious, when the player character is about to get in a dangerous spot. “Dark red stains have seeped into the floorboards around the door” in the room description would be a good way to warn the player that opening the door could be dangerous.

It’s all too easy to write a game, even without meaning to, in such a way that the player can make the game unwinnable. For instance, the player may need to give the chocolate biscuit to the elf so that the elf will be willing to part with the silver key … but the biscuit might be edible, and the player might have eaten it a hundred moves before meeting the elf. Another example: Maybe the player can’t return to a region of the map after leaving it. A rope bridge might have fallen the first time the player crossed it, leaving no way to get back across the canyon. If the player dropped the gold key on the far side of the bridge, she won’t be able to go back and get it. If the gold key is required to open something that’s on *this* side of the canyon, the game has become unwinnable.

You need to think carefully about where in your story this type of problem can arise, and decide how you want to handle it. Drawing a diagram of the puzzle structure may be helpful.

One way to handle the situation is to simply not allow the player to do anything that would make the game unwinnable. If the chocolate biscuit will be needed later, write something like this:

```
Instead of eating the chocolate biscuit, say "It's a mouth-watering treat, no doubt, but you decide to save it for later."
```

It can be a chore to come up with sensible-sounding reasons why the player character wouldn’t do something, if it seems a perfectly natural thing to want to do. After all, the player can’t read your mind, and doesn’t know what actions or objects will later be needed to win the game. For variety, I sometimes allow the player to do something that will turn out to make the game unwinnable, but then add an immediate message giving a broad hint (which the player is free to ignore) that that was probably a stupid thing to do:

```
After eating the chocolate biscuit:
	say "You chow down on the delicious biscuit and lick the last of the chocolate from your fingers. Afterward, though, you start to worry. Maybe you shouldn’t have indulged your gluttonous impulses quite so casually.";
	rule succeeds.
```

The player who reads this will probably be smart enough to UNDO the latest action, thereby retrieving the chocolate biscuit.

Keeping Score

Inform allows us to give the player points for doing certain actions. A way to do this is explained on **p. 9.2** of the Documentation, “Awarding points.” Normally, you would award points in an After rule, like this:

```
After taking the gold crown for the first time:
	award 10 points;
	say "Ah, you've finally got your hands on it!"
```

Unless we include “continue the action” at the end of the After rule, the After rule will halt the action processing *before* the Report rule has a chance to tell the player that taking the gold crown has succeeded. So if you’re not going to print out a message of your own when awarding points, you’ll want to add the line “continue the action” after awarding the points. This will cause the game to report “Taken”, “You put the bulb in the socket”, or whatever.

When awarding points, you should get in the habit of always including the phrase “for the first time” in the rule that awards the points. If you forget to do this, the player will be able to rack up a huge score by performing the same action over and over!

Having explained that, however, I’m now going to suggest that you not do it that way. The reason, as

Emily Short explained to me, is because the phrase “for the first time” applies only the first time the player *attempts* to do something, whether or not it succeeds. So if the gold crown is in a locked transparent display case and the player tries TAKE CROWN while the case is still locked, then later, when the case is unlocked, taking the crown will *not* award any points.

It’s much safer to use one of Inform’s built-in properties, **handled,** to check whether points should be awarded:

```
After taking the gold crown when the gold crown is not handled:
```

Use “for the first time” to award points *only* when you can be sure that the action will succeed the first time the player attempts it.

Examples 135 and 136 in the Documentation, “Mutt’s Adventure” and “No Place Like Home,” show other ways of awarding points. **Page 9.3**, “Introducing tables: rankings,” shows how to create a table of rankings that will tell the player how well he’s doing. This feature was popular in early IF games, and some authors still enjoy using it.

Inform doesn’t insist that the number of points awarded for an action be constant. If you like, you can do this:

```
crown-points is a number that varies. crown-points is 10.

After taking the gold crown when the gold crown is not handled:
        award crown-points points;
        say "Ah, you've finally got your hands on it!"
```

If you’ve set it up this way, you can vary the number of points the player will gain for taking the crown depending on what else has happened in the game. You can even award a negative number of points, thus reducing the player’s overall score. In my first game, “Not Just an Ordinary Ballerina” (which was written in Inform 6, not Inform 7), I set up a system that would reduce the number of points the player could gain by solving each puzzle based on the number of hints the player had consulted about that puzzle. The only way to get the maximum score for winning the game was never to consult the hints. This was meant to give players an incentive to use their ingenuity rather than relying on the hints.

The status line (at the top of the game window) normally shows the score the player has so far accumulated. If you put the line “Use no scoring” near the top of your Source, the score listing in the status line will be omitted, and awarding points in your Source will do nothing. In a game whose story is serious in tone, awarding points might seem too frivolous, so you might want to use no scoring.

If you do award points, it’s a good idea to keep a list somewhere of how many points are being awarded for each scored action. Add up the points. Near the top of your source code, tell Inform the maximum score, as suggested on **p. 9.2**, “Awarding Points”:

```
The maximum score is 12.
```

Note that you can’t say “The maximum score is 12 points.” The word “points” can’t be used here.

A Treasure Chest

Awarding points when an object is picked up or a room entered is the usual thing to do in a game. But some classic games require that the player put the objects that have been found in a treasure chest or trophy case. Awarding points for this action is just slightly tricky, because the treasure chest might be a closed openable container. If it happens to be closed the first time the player tries to put a given treasure into it, the points will never be awarded. Consider this code:

```
The Lab is a room.
The player is carrying a fish.
The old shipping trunk is a closed openable container in the Lab.

After inserting the fish into the trunk for the first time:
	award 1 point;
	continue the action.
```

That certainly looks as if it should work. But here’s the output:

```
>put fish in trunk
The old shipping trunk is closed.

>open trunk
You open the old shipping trunk.

>put fish in trunk
You put the fish into the old shipping trunk.

>score
You have so far scored 0 out of a possible 0, in 4 turns.
```

No points were awarded for putting the fish in the trunk, because the first time the player tried it, the trunk was closed. Here’s the correct way to write the After rule:

```
After inserting the fish into the trunk:
	if the fish is in the trunk for the first time:
		award 1 point;
	continue the action.
```

But wait ... what if we want the player to be awarded points only while the treasure *remains* in the treasure chest? This requires a slightly different approach. What we need to do is award a point each time the treasure is successfully inserted into the treasure chest, and also subtract a point each time the treasure is taken out of the chest. To subtract a point, just award -1 point. Here’s how to do that, using the example of the fish and the old trunk, as before:

```
After inserting the fish into the trunk:
	if the fish is in the trunk:
		award 1 point;
	continue the action.
```

```
Instead of taking the fish:
	if the fish is in the trunk:
		award -1 point;
	continue the action.
```

Achievements

Some authors feel that keeping a numerical score trivializes a serious game, or isn't interesting from a literary standpoint. Even so, it would be nice to let the player know what kind of progress he or she is making. Instead of producing a numerical score in response to the SCORE command, we can give the player a list of achievements. To do this, we'll use a table. Tables are one of Inform's more advanced features, and the syntax for using them is a little tricky. At the end of this example, so as to give you a better grasp of tables, we'll take a look at how the code works.

We're going to create a game with three achievements — picking up a ruby, taking off some goggles (which are being worn), and putting a guppy back in the fish tank. These are listed in the Table of Achievements. The table has three columns: object, achievement, and flag. In the object column, we'll enter the name of the object that is being handled when the achievement happens. In the achievement column is some text describing the achievement. The flag column contains a number; this starts out as 0, but we'll change it to 1 when the player accomplishes the achievement.

```
The Living Room is a room. The guppy is here. The ruby is here. The player wears some goggles.

The fish tank is an open container in the Living room.

After taking the ruby when the ruby is not handled:
	choose the row with an object of ruby in the Table of Achievements;
	now the flag entry is 1;
	continue the action.

After taking off the goggles:
	choose the row with an object of goggles in the Table of Achievements;
	now the flag entry is 1;
	continue the action.

After inserting the guppy into the fish tank:
	choose the row with an object of guppy in the Table of Achievements;
	now the flag entry is 1;
	continue the action.

Table of Achievements
object		achievement				flag
ruby		"picked up the ruby"			0
goggles		"removed the goggles"			0
guppy		"put the guppy back in the tank"	0
```

```
This is the new announce the score rule:
    let flag-count be 0;
    repeat with N running from 1 to the number of rows in the Table of Achievements:
        choose row N in the Table of Achievements;
        if the flag entry is 1:
            increase the flag-count by 1;
    if the flag-count is 0:
        say "You haven't done anything notable yet.";
    otherwise:
        let total-flag-count be flag-count;
        say "So far you have [run paragraph on]";
        repeat with N running from 1 to the number of rows in the Table of Achievements:
            choose row N in the Table of Achievements;
            if the flag entry is 1:
                say "[achievement entry][run paragraph on]";
                decrease the flag-count by 1;
                if the flag-count is 0:
                    say ".";
                    rule succeeds;
                otherwise if the flag-count is 1:
                    if the total-flag-count is 2:
                        say " and [run paragraph on]";
                    otherwise:
                        say ", and [run paragraph on]";
                otherwise:
                    say ", [run paragraph on]".
```

```
The new announce the score rule is listed instead of the announce the score rule in the carry out
requesting the score rulebook.
```

The heavy lifting in this example is done by the new announce the score rule, which replaces the announce the score rule (one of the Standard Rules). The new rule does two things. First, it uses a loop (“repeat with N running from 1...”) to count the number of achievements the player has so far done. Each time it finds a row where the flag is 1, it increases the flag-count. Then it prints out a message. If the flag-count is still zero, nothing has been accomplished, so the rule will say so and stop.

If the flag-count is at least 1, that means the player has done something. In this case we start the printout by saying, “So far you have [run paragraph on]”. Then we go through the Table of Achievements again, looking for and listing achievements. As we go, we decrease the flag-count — but we’ve taken the precaution of storing it in total-flag-count before the process starts. This is so we’ll know whether the list is exactly two items long, or whether it’s longer. If it’s exactly two items long, we want to print “ and ” between them, but if it’s three or more items long, we want to print “, and ” between the last two. This is the serial comma. If you’re not using the serial comma in your game (see p. 25, you can omit the lines that use total-flag-count.

For more ways to report scoring, see the examples on **p. 11.4**, “Scoring,” in the *Recipe Book.*

Chapter 8: Time & Scenes

Lots of interactive fiction games take place in a sort of "eternal now." The player is free to wander back and forth, trying whatever commands he thinks of. No matter how many turns he takes, nothing changes in the model world except when he takes an action that changes it.

In many other games, though, the passage of time has some effect on the game-play. That is, the game counts the number of turns the player spends doing things, and makes some change after a set number of turns. For instance, the sun might set, in which case outdoor rooms would become dark. In almost all games, nothing whatever happens between the player's commands: time passes only when the player does something. There are a very few games that include some sort of "real time" routine to force the player to type the correct command within a few seconds, but these games are quite rare, and we're not going to worry about them in this book.

Inform has three systems with which you can organize the passage of time of your game. In this chapter we'll look at all of them.

Passing Time

Inform doesn't do anything to mark the passage of time in a game unless you tell it to. All the same, it's quietly keeping track of the number of turns that have passed since the beginning of the game. Unless you write some code that changes the default, Inform starts the game at 9:00 A.M. Each time the player enters a sensible command, one minute passes. But if the player types something that the parser doesn't understand, no time passes.

A simple way to see this mechanism in action is to add this code to your game:

```
When play begins:
		change the right hand status line to "[score] / [time of day]".
```

The status line is the bar across the top of the game window. Normally (unless you do something to change it), the right side of the status line shows the number of points the player has earned and the number of turns that have passed since the start of the game. The code above changes the status line to show the score and the time of day. For more, see **p. 8.3** of the Documentation, “Changing the status line.”

As a courtesy to the player, you might want to have no time pass when the player is simply using the LOOK command to refresh her memory about the room, or the INV (or I) command to remind herself what she’s carrying. To do that, use this code:

```
Every turn:
        if the current action is taking inventory or the current action is looking:
                change the time of day to 1 minute before the time of day.
```

We’ll have more to say about Every Turn rules later in this chapter. Another way to do the same thing is to use a Before rule rather than an Every Turn rule:

```
Before looking or taking inventory:
        change the time of day to 1 minute before the time of day.
```

The downside of using a Before rule in this case is that the first action when a game starts is a look action. That’s what causes the initial room description to print. So if you use a Before rule in this case, the game will be starting one minute before you think it does, which could be a problem in a game that opens with a tight timing puzzle. (Thanks to Eric Eve for pointing this out.)

Example 391, “Timeless,” shows a different way to accomplish the same thing.

The game doesn’t have to start at 9:00 A.M. If you want it to start at 3:00 in the afternoon, do this:

```
When play begins:
     now the time of day is 3:00 pm.
```

If you want time to jump ahead for some reason (such as when the player character has been hit on the head and is unconscious for a while), include a line like this in your code as part of what happens when the player is hit:

```
        now the time of day is 7:00 pm.
```

If you want something to happen at a specific time, one way to do it — the primitive, clumsy way, as I’ll explain in a moment — is with an Every Turn rule:

```
Every turn:
        if the time of day is 11:59 pm or the time of day is 12:59 am or the time of day is 1:59 am:
                say "In the distance you hear the chimes in the tower strike the hour."
```

As you can see, I’ve set the chimes to ring one minute *before* the hour. This is because of the way

Inform runs your Every Turn rules. Every Turn rules run before the Advance Time rule. If you do it as shown above, the message about the chimes in the tower will be printed out on the hour.

In this case, Inform has a simpler way to get the same effect. You can write a rule that tells the game what to do at specific times:

```
At 11:59 pm: say chimes.
At 12:59 am: say chimes.
At 1:59 am: say chimes.

To say chimes: say "In the distance you hear the chimes in the tower strike the hour."
```

This way of doing it is much easier to read, especially when you want the chimes to ring every hour for 24 hours, rather than only at three times, as shown.

The main reason to want to keep track of the current time (and let the player know what time it is) is because time is passing in the model world. In some games, the player character will get hungry and/or thirsty on a regular schedule, so finding food and water will be one of the puzzles. If the game includes a realistic alternation of day and night, at a certain time in the evening the player might need to find a place to sleep. Some people consider eating and sleeping puzzles old-fashioned, but timed puzzles can be more interesting. For instance, in an outer space location, the player character might be wearing a space suit whose oxygen tank will run out after a set number of turns.

Future Events

Page 9.11 of the Documentation, “Future events,” shows how to cue up events that will happen at a certain time in the future. For the next example, I’ll borrow some code from the chapter of the *Handbook* on characters. When the player tries to take the spider, not only the game prevent the action, but there will be an after-effect — a sort of emotional echo of the intended action:

```
Instead of taking the spider:
	say "You can't bring yourself even to get near it.";
	the spider thought returns in four minutes from now.

At the time when the spider thought returns:
	say "You're still thinking about that creepy spider...."
```

Here I’ve created a new phrase, “the spider thought returns,” and told Inform what to do when that event happens. There are two ways to cue this action: You can say “in four minutes from now” or “in four turns from now” (substituting your own number of turns or minutes, obviously). If you’re making adjustments in the time of day, as shown earlier in this chapter in the section “Passing Time,” four minutes from now will *not* necessarily be the same as four turns from now.

If you need to trigger a more complex series of events rather than something that happens once, using Scenes (see below) will give you better tools.

If you want something to *maybe* happen in the future, depending on some factor, you can use Inform's handy "do nothing" phrase. The example below was suggested by one of my students, who wanted to transform a magic wand into an old shoe — but only if the player has been carrying the wand for three turns in a row.

```
The Wizard's Workshop is a room. "This odd-smelling little room is crowded with tables and shelves overflowing with magical implements."

The magic wand is in the Wizard's Workshop. "A Wham-O(tm) magic wand lies on the floor." The description is "It's a shiny black plastic wand on which the words 'Wham-O' are written in flowing white script." Understand "wham-o", "shiny", "black", and "plastic" as the magic wand.

The old shoe is a thing. The description is "It looks like somebody left it lying in the gutter."

After taking the magic wand:
	the wand transforms in three turns from now;
	continue the action.

At the time when the wand transforms:
	if the player does not carry the wand:
		do nothing;
	otherwise:
		remove the magic wand from play;
		now the player carries the old shoe;
		say "The wand quivers and squirms in your hand! Suddenly it's not a wand, it's an old shoe!"
```

Here, we start Inform's internal "alarm clock" ticking when the player takes the magic wand, using an After rule. If the player isn't carrying the wand when the "alarm clock" goes off, nothing happens. And if the player drops the wand and picks it up again, Inform will automatically reset the clock. The wand will only transform if it's carried for three turns in a row. Obviously, this type of scheduling of future events has many other uses.

Scenes

One of the more interesting features of Inform is its ability to organize time into scenes. An entire chapter of the Documentation (Chapter 10, "Scenes") is devoted to scenes. If you haven't read this chapter yet, give it a look. Some games won't need scenes at all, but scenes can be very handy for giving your story some structure. Almost any time you need to create a complex, well-coordinated set of changes in the model world, defining a scene is a good tool for the job.

To create a scene, you simply give it a name, and then tell Inform when it starts — that is, what set of events or circumstances triggers it. You can also, optionally, tell Inform when the scene ends. By default, a scene will happen only once. If you want it to be able to happen over and over, you need to create it as a *recurring* scene.

Why might you want to use a scene? Here are some random ideas:

```
Saucer-menace is a scene. Saucer-menace begins when the flying saucer is in the Meadow. Saucer-menace ends when the bug-eyed monster is dead.
```

```
Guard-evasion is a scene. Guard-evasion begins when the security guard is suspicious. Guard-evasion ends when the player is in the Bank Vault for the first time.
```

```
Dance mania is a scene. Dance mania begins when the player is in the Abandoned Warehouse for the first time. Dance mania ends when the police sergeant is in the Abandoned Warehouse.
```

When a certain scene starts, you might want to rearrange the objects in the model world, shuffling some items offstage and others onstage. While the scene is happening, you might want to restrict the player's travel, or print out certain atmospheric messages (as shown on **p. 10.4** of the Documentation, "During Scenes"). While a scene is happening, the magical weapons the player is carrying might have different effects than at other times. The possibilities are basically unlimited.

You can't start a scene simply by saying that it starts:

```
now guard-evasion is happening; [Error!]
```

Nor can you end a scene by saying that it ends. The way to start or end a scene is to tie it to a condition, as shown above:

```
Guard-evasion begins when the security guard is suspicious.
```

Chaining Scenes

Page 10.5 of the Documentation, "Linking scenes together," gives this simple example of how to chain two scenes:

```
Train Stop is a scene. Brief Encounter is a scene. Brief Encounter begins when Train Stop ends.
```

From this code, we can't tell what events will cause Train Stop to begin or end, but we can see that Brief Encounter will begin immediately when Train Stop ends.

Setting up a new scene so that it starts a certain number of turns after a previous scene ends is a little trickier. First, we need to create a truth state (a true-or-false variable). Second, we need to write a simple function that will switch the truth state to true. Third, we need to tell Inform to run the function at some time in the future. Here's an example that shows how to do it:

```
The Sidewalk Cafe is a room.
The Wine Shop is north of the Sidewalk Cafe.

Lunch-ready is a truth state that varies. Lunch-ready is false.

Breakfast is a scene. Breakfast begins when play begins. Breakfast ends when the player is in the Wine Shop for the first time.
```

```
When Breakfast ends: The gong sounds in three turns from now.

At the time when the gong sounds:
	now lunch-ready is true.

Lunch is a scene. Lunch begins when lunch-ready is true. When Lunch begins, say "Luncheon is served."

Test me with "scenes / n / z / z / z".
```

The phrase “The gong sounds” is entirely artificial. You could use anything here: There isn’t any gong in the game. It’s just a way of lining up an event to occur in the future. When the player enters the Wine Cellar, it causes the Breakfast scene to end. Three turns later, the Lunch scene begins.

We could just as easily skip the truth state and the delayed sounding of the gong like this:

```
Lunch begins when the player has been in the Wine Shop for three turns.
```

But if we do it that way, the player will have to remain in the Wine Shop for three turns in a row. If the player leaves the Wine Shop too soon, the Lunch scene won’t begin.

Using Scenes to Control Action

Here’s a rather convoluted but interesting example of how useful scenes can be. At the beginning of this little game, the player carries the crown jewels. If the player goes into the boudoir, where the princess is waiting, the princess won’t let the player leave until the jewels have been handed over. This situation is similar to the example on p. 176, in which Wyatt Earp demands that the player hand over his six-gun, but we’ll set it up using a different kind of code.

```
The Antechamber is a room. "The princess's boudoir lies to the south."

The Princess's Boudoir is south of the Antechamber. "This luxuriously appointed chamber is fit for a princess."

The princess is a woman in the Boudoir. "...and as it happens, a princess is sitting here right now!"

The player carries some crown jewels. The indefinite article of the crown jewels is "the".

Princess Demanding is a recurring scene. Princess Demanding begins when the player is in the Boudoir and the player carries the crown jewels.

Princess Demanding ends when the player does not carry the crown jewels.

Instead of going during Princess Demanding:
	say "'You're not going anywhere until you surrender the jewels!' the princess insists."

When Princess Demanding begins:
	say "'I notice you're carrying the crown jewels,' the princess says. 'Give them to me at once!'"
```

```
Instead of giving the jewels to the princess:
	now the princess carries the jewels;
	say "You surrender the jewels. The princess smiles sweetly. 'Thank you,' she says. 'You may go.'"
```

Here we’ve created a scene called Princess Demanding. The point of this scene is the “Instead of going” rule. This rule gets in the way if the player tries to leave the boudoir with the jewels. Other than this, the scene is transparent — that is, it has no effect on game-play. The player can drop the jewels or give them to the princess. At that point, the scene ends (because we’ve written a rule that defines this as ending the scene).

The keyword “during” in this Instead rule lets us write code that will make sweeping changes in many aspects of the game depending on whether a certain scene is happening.

We’ve made Princess Demanding a recurring scene. This is necessary. If you leave out the word “recurring,” the player can get away with the jewels by dropping them in the boudoir and then picking them up again. When they’re dropped, the scene ends — and if it’s not a recurring scene, it will never start again.

Let’s look at this scene more closely. It looks sensible on the surface, but in fact there are some problems. By analyzing the problems, you can start to get a better picture of how to write trouble-free code.

In a real game the player would quite likely be able to figure out a way to abscond with the jewels, even after the princess has noticed them. Can you spot the problem? If you’ve read the section in Chapter 3 on “Testing Where a Thing Is,” you may recall that the condition “if the player carries the crown” is only true if the crown is in the PC’s hands — that is, if it’s directly carried. The bug will show up if we give the player a container for carrying things:

```
The sack is a container. The player carries the sack.
```

Now the player can walk into the boudoir, which will cause the princess to notice the jewels, and then put the jewels in the sack and walk straight out again. Oops! The fix for this bug is simple. We change “carries” in the end-of-scene statement to “encloses”:

```
Princess Demanding ends when the player does not enclose the crown jewels.
```

If you make these changes, the player will be able to put the jewels in the sack and then enter the boudoir, and the princess will let him leave again. Presumably, she hasn’t noticed the jewels because they’re in the sack. But once the jewels are not in the sack but in the PC’s hands, she’ll see them. At that point, putting them back in the sack won’t help. The princess will still demand that they be turned over.

But there’s still a problem. Try this series of commands:

```
Test me with "s / put jewels in sack / drop sack / take sack / n".
```

The princess will notice the jewels ... but you can still get away with them by putting them in the sack, dropping the sack, and picking it up again, because the Princess Demanding scene won’t start again as long as the jewels are in the sack. Ah, but we don’t want to change the definition of the start of the scene so that it uses “encloses” rather than “carries,” because that would cause the princess to notice the jewels for the first time even if they’re in the sack. What a tangle!

The solution is in two parts. First, we need to keep track of what the princess knows. In addition, we’ll create a second scene, Princess Still Demanding. This scene will do the work of stopping the player from leaving — and it will have a more complicated beginning. Here’s the revised part of the code:

```
The princess is a woman in the Boudoir. "...and as it happens, a princess is sitting here right now!"
The princess can be jewel-knowing or jewel-ignorant. The princess is jewel-ignorant.

When Princess Demanding begins:
	now the princess is jewel-knowing.

The player carries some crown jewels. The indefinite article of the crown jewels is "the".

The sack is a container. The player carries the sack.

Princess Demanding is a scene. Princess Demanding begins when the player is in the Boudoir and
the player carries the crown jewels.

Princess Still Demanding is a recurring scene. Princess Still Demanding begins when the player is in
the boudoir and the player encloses the crown jewels and the princess is jewel-knowing.

Princess Demanding ends when Princess Still Demanding begins.

Princess Still Demanding ends when the player does not enclose the crown jewels.

Instead of going during Princess Still Demanding:
	say "'You're not going anywhere until you surrender the jewels!' the princess insists."
```

The first thing we’ve done here is give the princess a new either-or property. She can be jewel-knowing or jewel-ignorant. At the beginning of the game she’s jewel-ignorant. But when Princess Demanding begins, she becomes jewel-knowing. The moment she becomes jewel-knowing, the game switches to a different scene, Princess Still Demanding.

If you add this code to the original version of the example, the player will be able to go in and out of the boudoir freely carrying the jewels in the sack. But once the princess spies the jewels, the player will be prevented from leaving whether the jewels are directly carried or are being carried in a container.

There’s still at least one more problem with this example game, which illustrates just how tricky writing IF can be. If the player drops the jewels on the floor, the princess will just leave them lying there. She’ll let you leave, but if you leave the boudoir and come back, you’ll find that the jewels are still lying on the floor in plain sight. A writer of conventional fiction might describe this by saying that the princess’s motivation (her reasons for her actions) is not consistent. We can fix this with an Every Turn rule:

```
Every turn:
	if the holder of the crown jewels is the Boudoir:
		now the princess carries the crown jewels;
		say "The princess scoops up the jewels. 'These belong to my family,' she says, adding haughtily, 'you may go.'"
```

As an exercise, I'll leave you the chore of setting it up so that if the princess is jewel-knowing, the jewels are in the sack, and the sack is on the floor of the boudoir, the princess will take the jewels from the sack. The moral of the story is this: When planning an action, think about *all* of the actions a player might take. Think about all of the configurations the various objects might get into. Also, think about how any other characters in the room would naturally react when the PC does something or other.

Every Turn Rules

We've already seen a couple of examples of Every Turn rules. As you can guess from the name of the rule, an Every Turn rule is consulted every turn to see whether it would like to do anything. Note that this rule is consulted as the *last* step in the process that starts when the player types a command. The output of an Every Turn rule will normally appear at the bottom of a block of text in the game, after the description of what happens as a result of the player's latest command.

An Every Turn rule can print out some text, or it can do something more complicated. Printing out some text is a nice way to add atmosphere to your game, but a message that prints every turn will quickly become annoying. Here's an Every Turn rule that adds atmosphere to a forest setting in a slightly more interesting way. Sometimes it chooses a random text to print, and sometimes it does nothing:

```
Every turn:
	if a random chance of 1 in 3 succeeds:
		say "[one of]A dragonfly darts past you.[or]You hear a frog croaking.[or]A bird chirps in the bushes.[or]The breeze rustles the leaves.[at random]".
```

But even with a bit of randomness, this type of Every Turn rule will get boring before very long. A better solution is to write an Every Turn rule that only runs when a certain scene is happening, or when the player is in a certain region. If we have defined the Forest Area as a region, we could rewrite the rule above like this:

```
Every turn when in the Forest Area:
```

If we've created a scene called Forest Explorations, we could do it this way:

```
Every turn during Forest Explorations:
```

We can also write an Every Turn rule that will produce some narrative or background text at specific points within a scene. Here, the phrase "exactly three turns" will insure that the output text is produced only once (unless the scene is recurring, in which case it will be produced once per recurrence):

```
Every turn:
	if Saucer Menace has been happening for exactly three turns:
		say "You hear an odd glorping noise somewhere nearby.";
	otherwise if Saucer Menace has been happening for exactly five turns:
		say "Was that a tentacle you saw slithering out of sight?"
```

An Every Turn rule can also wait quietly in the background, checking for a certain set of conditions, and then do something when the conditions are met. Here’s a simple game that shows how the idea might work:

```
The Throne Room is a room. "The king's golden throne stands here."

The golden throne is an enterable scenery supporter in the Throne Room. The description is "A magnificent golden throne."

The jewelled sceptre is here. The sparkling crown is here. The crown is wearable.

Every turn:
	if the player wears the sparkling crown and the player carries the jewelled sceptre and the player is on the golden throne:
		say "You're the king!!!";
		end the game in victory.
```

Every turn, this rule checks to see whether the player has done all three things that are needed to win: the player has to be carrying the sceptre and wearing the crown while sitting on the throne. You could get the same results using After rules for three different actions (after taking the sceptre, after wearing the crown, and entering the throne), and test in each of the After rules for whether the other two conditions were satisfied — but using an Every Turn rule is less likely to lead to a bug, because you can test all three conditions in one place. And if you need to edit the code for some reason, you only need to edit it in one place; you don’t need to hunt for every spot in the code where that condition is being checked.

Here’s another way to get the same result:

```
Every turn when the player is on the golden throne:
	if the player wears the sparkling crown and the player carries the jewelled sceptre:
		say "You're the king!!!";
		end the game in victory.
```

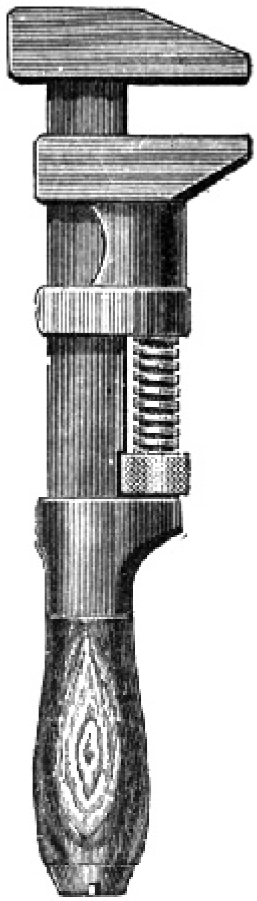
As this example shows, an Every Turn rule can include a test in its first line. If we’re writing a game in which the player needs to eat periodically, we might do this:

```
A person can be hungry or not hungry. A person is usually not hungry.
The player has a number called hunger-level. The hunger-level of the player is 0.
Every turn when the player is hungry:
	increase the hunger-level of the player by 1;
	say "[one of]Your stomach is rumbling.[or]You're becoming quite hungry.[or]You're very hungry. You need to find food soon.[or]You're practically starving![stopping]";
```

```
    if the hunger-level of the player is 7:
        end the game saying "You have starved to death!"
```

Switching the player to hungry would happen elsewhere in the code. In this example we’ve added something new — a counter (hunger-level) that keeps track of how long the player has been hungry, and ends the game after a set number of turns. Eating food would both switch the player to not hungry and reset the hunger-level of the player to 0.

Chapter 9: Phrasing & Punctuation



In this chapter we're going to look at the guidelines you'll need to follow when writing your game in the Inform language. Most programmers refer to what they write as "source code." The Inform Documentation refers to what you'll be writing as the "source text," but the only reason I can see for using this nonstandard term is to make Inform look less intimidating to the new author. What you'll be writing will be source code, and that's the term used in this book.

If you've ever learned a traditional computer programming language, you may be surprised when you first look at the syntax of the Inform language. ("Syntax" is just a fancy word for how things are phrased and punctuated.) At first glance, Inform looks very little like a programming language; it looks pretty much like plain written English.

This is an illusion — artful, but an illusion. Just like any other programming language, Inform has a compiler. And like any other compiler, the Inform compiler absolutely insists that your code be written in certain ways. The "natural language" user interface of Inform is not much more than skin-deep.

The use of "natural language" for programming is both a blessing and a curse. It's a blessing because if you don't remember how to do some type of task, you can try just writing it exactly the way you'd phrase it if you were speaking to a friend. Sometimes you'll find the right phrase that way. Or not. Many phrases that are understood without trouble by native speakers of English are *not* understood by Inform. Sometimes there are three or four different ways to write a sentence that will end up doing exactly the same thing, so you're pretty much guaranteed to hit the target before very long. At other times, syntax that you might naturally expect to work, won't, and you'll have to hunt around in the Documentation to find the correct syntax.

A second surprise, for expert programmers, may come from the sheer number of phrase constructions that are both allowed and needed by Inform. Inform is a language that's rich in features. As you write interactive fiction, you'll be dealing with innumerable specifics — odd connections between rooms, objects that need to behave in unexpected ways, in-game error messages that need to be customized due to the nature of the story, and on and on and on. Interactive fiction is a mass of particulars; generalizations are hard to come by, and even harder to nail down. But let's give it a shot.

Types of Source Text

There are three main kinds of writing in Inform. First, some of your writing will be between double quotation marks, "Like this." There are two types of things that you put between quotation marks: what you want your game to print out to the computer screen (at some point or other) when the game is played, and words and phrases that you want Inform to understand as input from the player. Here's a quick example that shows both of these types:

```
The will is on the end table. The description is "A legal-looking piece of paper. You can't read what's on
it, because it's upside down." Understand "legal", "legal-looking", "document", "paper", "piece", and
"piece of" as the will.
```

The sentence following "The description is" is output text, and the words after "Understand" are extra vocabulary words for the object whose main name is "will". The word "will" can be used both by the player of the game when referring to this object, and by the author. However, only the player can call it a "document" or a "piece of paper."

Words and sentences that are *not* between quotation marks but are instead surrounded by square brackets, [Like this], are *comments.* When your game is compiled, the comments will be ignored. Adding comments to your code is highly recommended! A comment is a memo you write to yourself, in order to take notes about things you still need to add to the story or so that you'll remember, six months from now, why your code is constructed in a certain way.

As explained below in the section "Text Insertions," however, square brackets have a completely different meaning when they're *inside* double-quotes.

Writing that's not between quotation marks and not surrounded by square brackets consists of instructions to the computer on how you want your game to operate — code, in other words.

With some minor exceptions, which we'll get to below, Inform does not care what you put between quotation marks. You can write amazing poetry, if you feel up to it, and become known as a writer of rare gifts. Or you can spell words wrong, use clumsy grammar, and indulge in sloppy punctuation. In the latter case, the people who play your game may think you're careless or stupid, but these mistakes won't matter to Inform. Inform won't even notice them. It will just say, "Oh, that's the text that I'm supposed to print out now." No problem.

Text that's neither a comment nor in quotes is a completely different matter. Inform will insist that you spell words correctly, that you follow certain rules for how phrases are worded, and that you use the right punctuation in every single line. Even a single mistake will stop Inform dead in its tracks. For instance, you might accidentally type a colon (:) instead of a semicolon (;) or vice-versa. This is easy to do, because they share a single computer key and look almost alike on the screen (especially when the screen is set to high resolution). But the colon and semicolon have completely different meanings to Inform. If you mix them up, when you click the Go! button to compile the game you'll see nothing but Problem messages. A list of a few common mistakes is given in Chapter 1 of this book.

In this chapter we'll look at the main rules you need to follow when writing your source code. It's not easy to set out a complete set of rules for you to follow, because there are so many different things you may want to do while writing interactive fiction. Inform has lots of features, some of which require special types of syntax.

The I7 Documentation gives fairly complete explanations of the syntax, but it's spread out in a lot of different places. Each type of syntax is discussed on the page where a feature that uses that syntax is introduced. In order to gather all of this information in one place, we'll have to refer to some Inform features that you may not have run into. Don't worry if you don't understand every detail: This chapter is partly for you to refer to when you're trying to figure out why Inform doesn't want to compile your game.

Text Insertions

It's extremely common for the game to need to make some decisions about what to print while the game is being played. The ways to do this are well described in **Chapter 5** of the Documentation, "Text," but we'll look at a couple of examples here.

In our first example, a ceramic bowl can be broken or unbroken. If it's broken, the description should obviously change. You can use square brackets to embed if/otherwise tests in the description of the bowl, like this:

```
The ceramic bowl is on the end table. The ceramic bowl can be broken or unbroken. The description is
"The delicate ceramic bowl is [if broken]chipped and cracked[otherwise]a beautiful work of art[end if]."
Understand "delicate" as the ceramic bowl. Understand "chipped" and "cracked" as the ceramic bowl
when the bowl is broken.
```

I've also created an Understand sentence that adds "chipped" and "cracked" as vocabulary — but only when the bowl is broken. The main point of this example, though, is the insertion of "[if broken]", "[otherwise]", and "[end if]" in the description. Notice that the period that ends the sentence is *after* the [end if]. The period will always print out, no matter whether the bowl is broken or unbroken. This is the way I prefer to write if/otherwise/end if blocks in my games, but you can also do it this way:

```
The description is "The delicate ceramic bowl is [if broken]chipped and cracked.[otherwise]a beautiful
work of art.[end if]".
```

Here, I've moved the period inside the if/otherwise text. So I also have to add a separate period *after* the close-quote, to let Inform know that the description sentence is finished. The advantage of doing it the first way is that the period just before the close-quote does double duty: Inform both prints it (because it's part of the output text) and understands it as ending the sentence that begins "The description is".

When a period, question mark, or exclamation point falls at the very end of a text block, just before the closing quotation mark, Inform understands that the punctuation mark both ends the sentence within quotes *and* ends the larger context of the surrounding code. For example, here's an error:

```
Instead of ringing the bell:
	say "The bell produces only a dull thud."
	now the butler is suspicious. [Error!]
```

The problem here is that the line beginning with "say" ends with a period. As a result, Inform thinks the Instead rule is finished. If we need to continue with more lines of code after a close-quote like this, we need to tack on a semicolon to tell the compiler that we’re not done yet:

```
Instead of ringing the bell:
    say "The bell produces only a dull thud.";
    now the butler is suspicious. [Now the code will compile, because we added a semicolon.]
```

But we were talking about text insertions, not about periods. You can string together several tests within a single block of output text by using “[otherwise if ...]”, like this:

```
The ceramic bowl can be intact, chipped, or shattered. The description is "The delicate ceramic bowl is
[if intact]a beautiful work of art[otherwise if chipped]slightly damaged[otherwise]shattered into
shards[end if]."
```

You’ll notice that we don’t have to tell Inform *what* object we’re asking “if intact” about. Inform knows that, by default, the object being examined is the one whose properties are being tested. We could test something entirely different, though, if we wanted to:

```
The description is "[if the curator is in the location]The bowl looks quite valuable, but you don’t dare
get close enough to give it a good inspection, not while the curator is hovering nearby, looking
nervous[otherwise]The bowl is a Ming Dynasty vase, clearly worth millions[end if]."
```

What we can’t do is embed one if-test inside another. This type of thing won’t work:

```
The description is "[if intact][if the curator is in the location]You don't want to appear too interested in
the bowl while the curator is hovering about[end if]The bowl appears to be a priceless Ming Dynasty
vase[otherwise if the bowl is chipped][if the curator is in the location] ...". [Error!]
```

When you need to test several conditions at once while assembling an output text, the most convenient and reliable way to do it is to use a To Say statement. We might write a To Say statement for our ceramic bowl like this:

```
The ceramic bowl is on the end table. The ceramic bowl can be intact, chipped, or shattered. The
description is "[bowl-desc]."

To say bowl-desc:
    if the bowl is intact:
        if the curator is in the location:
            say "You don't want to appear too interested in the bowl while the curator is
hovering about";
        otherwise:
            say "The bowl appears to be a priceless Ming Dynasty vase";
    otherwise if the bowl is chipped:
        say "The bowl is slightly chipped";
        if the curator is in the location:
            say ", as you know only too well, having chipped it yourself. The curator is
guarding it jealously";
    otherwise:
        say "The bowl is shattered beyond repair".
```

Notice the slightly unusual way the "say" lines are formatted. None of them ends with a period inside the close-quote. This is because the period that ends the description is up above, in the description property for the ceramic bowl. When you use a To Say rule in this way, you need to be especially careful about where the period is that ends the last sentence. This is because Inform does some special output formatting when it hits the end of a quoted sentence or paragraph. If the period is in the wrong place, a blank line can disappear from the output, or an extra blank line can be added.

Extra & Missing Blank Lines in Output

A frequent problem for beginning Inform authors is that the game output has either too many blank lines in it, or not enough of them. The biggest cause of this is periods not being placed correctly at the ends of double-quoted text blocks. When you start inserting if-tests and alternate blocks into your text, this problem is more likely to show up.

You can put any word you want in double-quoted text using brackets, as shown above, and then write a To Say statement that will run when the bracketed word is reached. Hyphenated compound words, such as "bowl-desc", are a good choice, because they're unlikely to be used anywhere else in your code.

Normally a To Say statement should always be called from within text that's going to be printed out in the game. This type of thing is legal, but not a good idea:

```
After attacking the ceramic bowl:
        say "[another-code-block]".

To say another-code-block:
        if the ceramic bowl is shattered:
                [more code goes here...]
```

Instead of using "say", a better way to call a separate code block, if you need to do it, is just to create it using a To statement, like this:

```
After attacking the ceramic bowl:
        sound the alarm.

To sound the alarm:
        if the ceramic bowl is shattered:
                [more code goes here...]
```

But there's no requirement that the To Say rule actually print any text to the game's output. Once in a while you may want to cause something to happen as part of a text output. Here's a simple example — we're going to adjust this NPC's mood when you ask him to give you an object:

```
Alexander Button can be friendly, neutral, or hostile. Alexander Button is neutral.

Instead of asking Alexander Button for the will:
	say "[Alexander-unfriendly]Alexander laughs at you."

To say Alexander-unfriendly:
	if Alexander is friendly:
		now Alexander is neutral;
	otherwise if Alexander is neutral:
		now Alexander is hostile.
```

In this example, the text output "Alexander laughs at you" includes a call to a To Say rule that prints nothing. It has only one effect: it causes Alexander to become less friendly by adjusting one of his properties. In this case, you could get the same result using a To phrase, as shown earlier in this section, by adding a line like "crank up Alexander's hostility" in the Instead rule — but there are times (such as when printing text that's stored in a table) when using a To Say statement is really the best way to do it.

Quotes Within Quotes

When you start writing NPC conversations, it's easy to forget that the quotation marks surrounding your text won't be printed out. This, for instance, is an error:

```
Instead of asking Alexander Button for the will:
	say "I would never give it to you!"
```

Here's the output:

```
>ask alexander for will
I would never give it to you!
```

This reads like a response from the parser, not from Alexander. The correct way to do it is using single quotes within the double quotes, like this:

```
Instead of asking Alexander Button for the will:
	say "'I would never give it to you!'"
```

This produces a better output:

```
>ask alexander for will
"I would never give it to you!"
```

Now the player can see that it's Alexander talking. A better writing style would be to include a *dialog tag* in the text:

```
Instead of asking Alexander Button for the will:
	say "'I would never give it to you!' he replies with a sneer."
```

As explained on **p. 2.3** of the Documentation, "Punctuation," Inform will automatically turn single quotes into double quotes when formatting its output. But this raises a potential problem — how to handle apostrophes. If an apostrophe is in the middle of a word, Inform will understand that it's an apostrophe, and won't turn it into a double quote when outputting the text:

```
Instead of asking Alexander Button for the will:
        say "'I'd never give it to you!' he replies with a sneer."
```

But if the apostrophe is at the end of a word, as in some types of dialect and informal speech, Inform will get confused:

```
Instead of asking Alexander Button for the will:
        say "'I'm not givin' it to you!' he replies with a sneer." [Error!]
```

To get an apostrophe into the output at the end of the word givin’, we have to put brackets around it:

```
Instead of asking Alexander Button for the will:
        say "'I'm not givin['] it to you!' he replies with a sneer."
```

Breaking Up Paragraphs

If you write a description (most likely, a room description) that runs on for several paragraphs, you can separate the paragraphs within a single double-quoted block by using two Return/Enter key presses, like this:

```
The Desolate Moor is a room. "A gloomy treeless waste stretches out on all sides. A few rocky
outcrops add an air of ancient menace.

Closer to hand, a crumbling castle stands."
```

When the game is played, there will be a standard paragraph break (that is, a blank line) before the sentence, “Closer to hand....”

Another way to get the same effect is with the text insertion “[paragraph break]”:

```
The Desolate Moor is a room. "A gloomy treeless waste stretches out on all sides. A few rocky
outcrops add an air of ancient menace.[paragraph break]Closer to hand, a crumbling castle stands."
```

This is a better, safer way to break up multiple paragraphs when the text appears in tables and conversation responses. Note that there is no space after “break]”. We want the next paragraph to begin at the left margin. A space character in quoted text will always be printed.

Occasionally you may need to go the other direction. You may need to write two separate “say” statements and have them printed together, without a paragraph break. In this case, the tool of choice is to end the first with “[run paragraph on]”. Here’s a handy little utility that illustrates the use of “[run paragraph on]”: If the player picks up something without having examined it, the description is automatically tacked onto the output after the word “Taken.”

```
A thing can be examined or unexamined.
After taking something unexamined:
	say "Taken. [run paragraph on]";
	try examining the noun.
Carry out examining something:
	now the noun is examined.
```

This code creates a new property of all things, and sets the property whenever a thing is examined. But the point of the example is this: If "[run paragraph on]" isn't included, the description of the thing that's picked up will be on a separate line. It will look more natural if it follows immediately after the word "Taken." is printed. Try it both ways to see the results for yourself.

Large Text Blocks

Inform has an upper limit on how much text can be included in a single block. I'm not sure what the limit is, but I've run past it a few times — for instance when adding a long block of text to the game that will tell newcomers to IF how to play the game. The compiler's Problem message says, "Too much text in quotation marks." It also says, "The maximum length is very high, so this is more likely to be because a close quotation mark was forgotten." Maybe yes, maybe no.

The solution is to create an object and give it several properties that are all blocks of text. Here's the intro of my game "A Flustered Duck" — not the intro itself, but the structure I created to print it out:

```
When play begins: say "[intro-1 of the text-holder][intro-2 of the text-holder][intro-3 of the text-holder]".

The text-holder is a thing. The text-holder has some text called intro-1. The text-holder has some text called intro-2. The text-holder has some text called intro-3.

intro-1 of the text-holder is "'Elliott! Elliott! Where are you, boy?' Granny Grabby's screeching voice rouses you from your pleasant, drowsy contemplation of a moth that has blundered into the barn and is flitting aimlessly about, trying to find its way back outdoors. [...and so on...]
```

White Space

When you're writing a game in Inform, does white space in your code matter? Sometimes yes, sometimes no. Because Inform is designed to look like "natural language," its use of white space is a little trickier than in some other programming languages.

Places where white space matters include around headings (see "Headings," later in this chapter) and when you're using Tab characters to organize tables and logical blocks of code (see "Indentation").

Each heading must have a blank line before it and a blank line after it. The line containing the story title and byline (on the first line of the Source file) must also have a blank line after it. The items within a table row must be separated by at least one Tab character, but you can add extra Tabs or more spaces if you like; they'll just be ignored.

When copying source code to or from an email or newsreader program, you need to be aware that some programs of this sort automatically turn Tabs into rows of spaces. The text will look the same on the screen — but if Inform is expecting a Tab, it won't understand the text anymore!

White space is preserved in double-quoted text when it is output to the screen during the game, so when you start adding conditional code within double-quoted text, you need to be careful to put spaces only where you want them. For instance, this is a text formatting mistake:

```
The description is "The bowl is [if intact] a priceless treasure [otherwise] badly damaged [end if]." [Mistake!]
```

Can you see what will happen? It looks easier to read in the source code, because we've separated the [] blocks by putting spaces on both sides … but when the text is printed out in the game, there will be two spaces after "is" and a space before the final period. Technically, we might not call this a bug, since your game will run perfectly, but there will be extra spaces in the sentence that appears in the game.

In writing source code, you definitely need to put white space around words. Borrowing from the example in the section on "Text Insertions," above, this will work:

```
To say bowl-desc:
```

...but this obviously won't:

```
Tosaybowl-desc:
```

There's no penalty for adding extra spaces, however. This will work, though there's no reason to do it:

```
To    say  bowl-desc    :
```

In some situations, you can insert one extra carriage return, but if there's an empty line, Inform won't like it. This will work:

```
To say
bowl-desc:
```

...but this won't:

```
To say

bowl-desc:
```

In many other situations, white space doesn't matter. As far as Inform is concerned, each sentence (that is, each block of code that ends with a period) stands on its own. You don't even need to put a space after the period. And if the sentence is followed by a blank line, you don't need to end the sentence with a period either. So these five ways of writing a pair of sentences are exactly the same, and all of them will work:

```
Painting is an action applying to nothing.Understand "paint" as painting. [No space after period]
```

or:

```
Painting is an action applying to nothing. Understand "paint" as painting. [Space after period.]
```

or:

```
Painting is an action applying to nothing.
Understand "paint" as painting.
```

or:

```
Painting is an action applying to nothing.

Understand "paint" as painting.
```

or:

```
Painting is an action applying to nothing

Understand "paint" as painting  [No periods.]
```

But this won't:

```
Painting is an action applying to nothing
Understand "paint" as painting
```

If Inform doesn't see a period and doesn't see a blank line either, it can't compile the code.

Blank lines can't be used within code blocks, however. This won't compile:

```
Instead of looking when the location is the hall of mirrors:

        if the player is invisible:

                say "It feels weird to look in the mirrors and not see your
handsome features reflected back." [Errors!]
```

Objects vs. Rules

Many computer programming languages are firmly based on the idea of objects. Even if you're creating something as abstract as an email program, every button or icon on the screen and every window that opens will probably be a separate object within the software. The technical definition of "object" doesn't matter at the moment: You can just think of a software object as a bunch of related code.

As you write with Inform, you'll be creating lots of software objects. Usually these will correspond to objects in the model world. A tree, for instance, would almost always be modeled using an object. In fact, it would be a *thing,* which is a kind of object. But Inform is unusual in that it isn't as object-oriented as many programming languages. If you have some experience programming in another language, you may be dismayed at first to find that many types of data can be created either as free-floating global values or as properties of objects.

It really doesn't matter which type of data you use, because all of the data properties of objects are public. They have global scope. (If you've never done any object-oriented programming, this paragraph will make no sense to you. Feel free to ignore it.) Also, objects in Inform don't have methods. All functions are global, although they can be written in such a way as to apply only to one object.

Instead of calling methods on objects, Inform uses rules and rulebooks. When you write rules in Inform, you can put them pretty much wherever you'd like in the source. When your game is compiled, Inform will collect all of the rules you've written and all of the rules in the Extensions you've included, and assemble them all into rulebooks.

When a rulebook is consulted during gameplay, Inform will proceed downward through the list of rules in the rulebook. When it finds a rule that applies to the current action, it will follow the rule. If the rule makes a decision (that is, if it ends with "rule succeeds" or "rule fails"), Inform will stop. Otherwise, it will continue on through the rulebook.

The rulebooks are constructed by the compiler according to certain principles. If you're curious how this works, you should definitely read **Chapter 18** of the Documentation, "Rulebooks." Basically, rules that are more specific will be listed earlier in the rulebook, while more general rules will be listed later. For instance, a game might have these two rules:

```
Instead of inserting something into an open container:
	[...more code would go here...]

Instead of inserting something into the coffee cup:
	[...more code...]
```

When Inform constructs the Instead rulebook, it will put the rule about inserting into the coffee cup *before* the rule about inserting into an open container, because the rule about the coffee cup is more specific — it relates only to one object, not to any container that has the open property. When the game is being played, the Instead rule for the coffee cup will be consulted first. It will (presumably) end with a default "rule fails", so action processing will halt. The rule about inserting something into an open container will never be reached.

As **p. 18.4** of the Documentation ("Listing rules explicitly") explains, we can tinker with the ordering of the rules if we need to. We can tell Inform to put a rule first or last in a rulebook. If a rule has a name (and most of the rules in Inform's Standard Rules have names), we can unlist them, like this:

```
The can't exceed carrying capacity rule is not listed in any rulebook.
```

Or we can replace an existing rule with our own named rule, like this:

```
The new carrying capacity rule is listed instead of the can't exceed carrying capacity rule in the check
```

```
taking rulebook.
```

If you need to figure out which rules (either the rules in the Inform library, or rules in an Extension, or new rules that you've written) are causing a certain output in a situation in your game, use the RULES command and then inspect what happens when you give the command that causes that output. You can download a file containing Inform's Standard Rules from the Inform website. You can also read the Standard Rules using the Open Extension command in Inform's File menu — but whatever you do, *don't edit this file!* The details of Inform's Standard Rules have been worked out through years of deep thought, trial-and-error, and highly technical bug reports from authors. Even small changes will quite likely cause bad things to happen.

The only reason to open the Standard Rules, other than simple curiosity, would be to copy a rule, insert it in your code, give it a new name, edit it in your code, and then replace the old rule with your newly edited version using code like the line shown above. Doing this is safe, and it's occasionally necessary, but it's not something to try until you've learned a lot about Inform — and even then, it's not something to do casually.

Unfortunately for those who are curious about how the Standard Rules operate, some of them simply refer to underlying code in Inform 6, which is not to be found in the Standard Rules. The manner in which the Inform 7 compiler uses Inform 6 code is not something we can reasonably get into in this *Handbook*, as it's probably the most advanced topic in Inform programming.

Starting with release 5Z71 of Inform, the Rules tab in the Index has some very nice tools for looking at the rules in your game. You can even remove many of the Standard Rules by clicking a single button, which will insert a line of code into your game.

The Names of Things

When you create a thing in your model world, the name you give it is understood by both the compiler and the parser. Most of the time, this is a convenient feature. But once in a while you may want to override it. You can do this using the **privately-named** and **printed name** properties. If you've read Chapter 2 of this *Handbook*, you may recall that we looked at a problem that can crop up if you try to name a room using a direction word. If you have a room called Hut, for instance, creating another room called South of the Hut is possible, as explained on **p. 3.2** of the Documentation, "Rooms and the map," but it's awkward.

I prefer to do it this way:

```
South of the Hut is Hut-South-Side. The printed name of Hut-South-Side is "South of the Hut".
```

This will work nicely, but if you do it this way, you'll have to call the room Hut-South-Side each and every time you mention it in your code.

While we're at it, we'll put a thing in this room:

```
The object22 is a privately-named thing in Hut-South-Side. The description is "It's a beautiful little yellow flower." Understand "beautiful", "little", "yellow", and "flower" as object22. The printed name of object22 is "a yellow flower".
```

This object can’t be referred to by the player as “object22”, because it’s privately-named. In this case, there would be no reason not to simply call the object “yellow flower” in your own code. But in some situations, creating a privately-named object might be useful. If you do so, remember to also give it a printed name.

As **p. 4.10** of the Documentation “Conditions of things”) shows, you can add variable output to a printed name. Here are two ways to do it, either of which might be useful in some situations:

```
The printed name of the ceramic bowl is "[bowl-condition] ceramic bowl".
```

```
The printed name of the ceramic bowl is "[if broken]broken [end if]ceramic bowl".
```

Punctuation

Inform is generally fussy about punctuation, but you have some options. In rules that have only one line, you can use either a comma or the word “then”. These two examples both work, and they do the same thing:

```
Instead of taking the ceramic bowl:
        if the bowl is shattered then say "What would be the point?";
        otherwise continue the action.
```

```
Instead of taking the ceramic bowl:
        if the bowl is shattered, say "What would be the point?";
        otherwise continue the action.
```

We can format this same code in some other ways too:

```
Instead of taking the ceramic bowl:
        if the bowl is shattered:
                say "What would be the point?";
        otherwise:
                continue the action.
```

```
Instead of taking the ceramic bowl:
if the bowl is shattered begin;
say "What would be the point?";
otherwise;
continue the action;
end if.
```

The fourth example is organized using Inform’s semicolon syntax, which allows indents using Tabs, but doesn’t require them. (For more on this syntax, see the section on “Indenting,” later in this chapter.)

Note that if we're using the semicolon syntax, there is *no* comma after "if the bowl is shattered".

If we're using the semicolon syntax, as shown immediately above, Inform not only doesn't care about indents, it doesn't care whether we include carriage returns at all. We could just as easily write it this way (though it's much harder for humans to read):

```
Instead of taking the ceramic bowl: if the bowl is shattered begin; say "What would be the point?";
otherwise; continue the action; end if.
```

Of the five examples above, I prefer always to use the format in the third one. When I use a consistent format, it's easier for me to spot mistakes.

Assertions

As explained on **p. 2.1** of the Documentation, "Creating the world," an *assertion* is a statement about something — usually about something in the model world you're creating. A lot of your writing in Inform will consist of assertions. Here are some assertions:

```
A saucer is on the table. The saucer is a supporter. A teacup is on the saucer.
```

```
North of the Sandy Beach is the Rocky Cove.
```

```
David is a man in the Living Room. David wears a leather vest and carries a walking stick.
```

```
Eye-color is a kind of value. A person has an eye-color. The eye-colors are blue, brown, and green.
```

The phrase "a kind of value" has a special meaning to Inform. We'll discuss that in the section "Values," below. The reason to include it here is so you'll see that some assertions are a little more abstract than others — they may be about forms of computer data that Inform will need to know, rather than about physical objects in the model world.

The first thing to notice about assertions is that each assertion is a complete sentence. You don't actually have to begin your sentences with capital letters — Inform doesn't care about this. But it's a good idea to get in the habit of writing Inform code so that it looks more or less like normal English. (If you're in the habit of texting in lower-case, using capitals may seem weird, but most forms of published writing require capitals.) An assertion can end either with a period or with a blank line of white space.

The second thing to notice is that the verbs in assertions are in present tense. ("Tense" is a term that refers to whether the action in the sentence takes place in the past, present, or future.) We can't do this:

```
David was a man in the Living Room. David wore a leather vest and carried a walking stick. [Errors!]
```

Those verbs are in past tense, and Inform won't understand them. Your printed output can be in the past tense if you like, but that requires some extra programming. It's a complicated subject that we're not going to get into in this book.

The third thing to notice is that when we're writing assertions about physical objects, it's a very good idea to start the sentence with “A,” “An,” or “The” in the normal way. (These words are called *articles.*) With some types of phrases, Inform won't care whether you use articles or not. But when you're first creating an object, Inform will notice whether you use an article. It will also notice whether you begin the name of the object with a capital letter.

Here are three assertions that are very similar except for their use of articles and capital letters:

```
The spoon is on the table.

Knife is on the table.

A Plate is on the table.
```

If we include these assertions in our game (after adding a table to the game, obviously), Inform will assume that Knife should always be capitalized and should never be given an article. It will assume that Plate should also be capitalized, but that an article should be used — because that's what the assertions did. When the game gets around to constructing a list of the items on the table (Inform will do this automatically in certain situations), the game will print out the list like this:

```
On the table are a spoon, Knife, and a Plate.
```

This is ugly. But far from being a defect in Inform, this behavior is a strong point. If you read **p. 3.18** in the Documentation, “Articles and proper names,” you'll see how easy it is to create objects whose names will be printed out in various sensible ways. The moral of the story is simple: Inform 7 can do almost anything that you might want it to do — but it's up to you to think carefully about what you want it to do!

Values

Values in Inform are pretty much like what other computer programming languages call *variables*. And in fact, when we create a value we can either call it a variable, or say that it varies:

```
X is a number variable.
Y is a number that varies.
```

But as **pages 4.7** (“New value properties”) through **4.12** (“Values can have properties too”) of the Documentation show, most of the values authors use in Inform programming are not numbers but words or even blocks of text:

```
X is a text variable. X is "Ugh."
Y is a text that varies. Y is "Wow!"
```

After creating the values above (X and Y), we can *initialize* them if we like by telling Inform what to store in the value at the start of the game. The statements “X is “Ugh.”” and “Y is “Wow!”” should be read as starting with an invisible “When play begins:”. They’re not permanent assignments — the data stored in X and Y can change during the game. What we can’t do is change the *kind* of data stored in the value. Elsewhere in our code, we could say “now X is “Okay, I guess....””, but we couldn’t say “now X is 17”, because that would attempt to turn X from a text into a number.

What’s extremely interesting and useful for game design is that Inform’s word values can have properties. If you look at **p. 4.12**, you’ll see these two statements:

```
Brightness is a kind of value. The brightnesses are dim, shining, and blazing.
A brightness can be adequate or inadequate. A brightness is usually adequate. Dim is inadequate.
```

Why would you want to do this? Because now you can test whether the brightness of a thing is adequate:

```
A lamp is a kind of thing. A lamp has a brightness. The brightness of a lamp is usually shining. The table lamp is a lamp.
```

Now we can write a test such as, “if the brightness of the table lamp is adequate”.

Values that are terms can be incremented or decremented just as if they were numbers. We’re allowed to do this:

```
increase the brightness of the table lamp by 1;
```

But we need to be careful about it, because Inform will let the value run off the cliff at the end. If the brightness (as defined in the code above) is blazing and we increase it without first checking to see if it’s already blazing, the next time the game tries to print out the brightness, it will say “<illegal value>”. To prevent this, we would need to do something along these lines:

```
if the brightness of the table lamp is not blazing:
	increase the brightness of the table lamp by 1;
```

In addition to variable values, Inform includes what programmers call *boolean* values. A boolean variable can have only one of two possible values: It’s either true or false. Inform calls this type of variable a **truth state**. Often it’s easier to use named value properties rather than truth states, but sometimes a truth state will do the job more succinctly. Here are two ways to accomplish pretty much the same thing:

```
The ceramic bowl can be broken or unbroken. The ceramic bowl is unbroken.

The ceramic bowl has a truth state called brokenness. The brokenness of the ceramic bowl is false.
```

If we do it the first way, we can later test “if the ceramic bowl is broken”. If we do it the second way, we need to test “if the brokenness of the ceramic bowl is true”. That’s a slightly more cumbersome way to phrase the code.

In some programming languages, you could write, “if the brokenness of the ceramic bowl:”, and the language would understand that because brokenness has to be true or false, this if-test is sensible, and should be evaluated as either true or false. But Inform won't do this. We have to write, "if the brokenness of the ceramic bowl is true:".

More information on values is found on **p. 8.1** and **p. 8.4** ("Change of values that vary" and "Change of either/or properties") of the Documentation.

When you create a value and attach it to an object as a property of that object, you should always tell Inform what value to give the property at the start of the game. This is called *initializing* the value:

```
The drill bit is a thing. The drill bit has a number called length. The length of the drill bit is 6.
```

(If you want to have the length be in inches, centimeters, or some other unit of measurement rather than just a number, consult **p. 14.3** of the Documentation, "Units.") You don't actually have to initialize the value of a number, but if you don't, your game may not work the way you expect it to.

If you're creating a kind of object — something that you'll have several of in your game — you can give every instance of the kind the same property. Again, initializing the value is a good idea:

```
A drill bit is a kind of thing. A drill bit has a number called length. The length of a drill bit is 6.

The iron bit is a drill bit. The length of the iron bit is 4.
```

Another way to do this is to create a kind of value and then give the object that kind of value:

```
Hardness is a kind of value. The hardnesses are hard and soft.
A drill bit is a kind of thing. A drill bit has a hardness. The hardness of a drill bit is hard.
```

This doesn't prevent you from creating some particular drill bit with a hardness of soft; all the code above is really saying is that the hardness of a drill bit is *usually* hard (unless you happen to write something that changes it). Experienced Inform programmers use the word "usually" in this situation:

```
The hardness of a drill bit is usually hard.
```

In the current version of Inform (5Z71), "usually" doesn't do anything, but this may be a bug. In future releases it may change. So it's a good idea to get in the habit of saying "usually" with values that may need to change during gameplay. Likewise, you can say "The hardness of a drill bit is always hard." But "always" is meaningless; it doesn't produce a constant value. The hardness can still be altered by another line in your game's code.

Temporary, local values can be created within code blocks. If you're new to programming, you may not realize that these temporary values have meaning *only* within the code block where they're defined. Once that block finishes, the value is thrown away. For instance, here's some code borrowed from Chapter 5 of this book:

```
After reading a command:
        let T be indexed text;
        let T be the player's command;
        [... and so on ...]
```

The word “let” is used to create a temporary value (that is, a variable). Here, we’ve created a temporary value called T. In this situation, we don’t need to tell Inform that T is a value that varies; Inform understands that it will probably need to vary while the After rule is running. What you need to know is that T can’t be referred to anywhere else in your code. It exists only within this After rule. In fact, you can write many different rules that have values with the same name, such as “X” or “the nearby room”. This doesn’t produce a bug.

Within the code block, you can manipulate the temporary value in whatever way you might like — but your manipulations will have an effect on the rest of the model world only if you write some code that causes them to. If you need to store a temporary value that you’ve created within a code block in order to be able to use it later on in some other part of the program, you can do so by creating a permanent variable as a storage space:

```
The drill bit has a number called length. The length of the drill bit is 4.
Instead of drilling the hole:
    let L be a number;
    let L be the length of the drill bit;
    [...some other code here changes the temporary value L...]
    now the length of the drill bit is L.
```

Here’s an interesting example (adapted from a question posted on the newsgroup) of what can happen if you forget that a “let” value in a code block is temporary. The person who posted the question wanted to figure out how much money the player had in her wallet; and if there was enough money, to pay out some of it. Can you spot the bug in this code?

```
A dollar bill is a kind of thing. The description of a dollar bill is "It's an ordinary greenback." Understand "dollars" as the plural of dollar bill. Understand "greenback" as a dollar bill.

The player carries a wallet. The wallet is an open container. In the wallet are 20 dollar bills. The description of the wallet is "Leather, and rather the worse for wear."

Bribing is an action applying to one thing and requiring light. Understand "bribe [someone]" and "pay off [someone]" as bribing.

Instead of bribing the inspector:
    if the player carries the wallet:
        let d be the number of dollar bills contained in the wallet;
        if d is less than 10:
            say "You don't have enough cash to bribe the inspector.";
        otherwise:
            change d to d minus 10;
            say "After bribing the inspector, you have [number of dollar bills contained in the wallet] dollars remaining."
```

The output is this:

```
>bribe inspector
After bribing the inspector, you have 20 dollars remaining.
```

We’re told the inspector has been bribed, but no money has actually left the wallet. The problem is that the code only manipulates the value of d, which is a temporary local variable. It never actually removes the dollar bill objects from the wallet. But in fact the problems with this code run a bit deeper. What if the player has $20 available, but has already removed $12 from the wallet, leaving only $8 in the wallet? In that case, the Instead rule will fail when it ought to succeed. And what if the player has $7 in hand and $8 in the wallet? We still need to extract $10, but we don’t initially know where the bills are located.

We need to revise the Instead rule rather extensively. Using a little literary license just for fun, we might come up with the code below, which works as desired. Note that the actual handling of the dollar bills is done in repeat loops. For more on loops, see p. 239.

```
Instead of bribing the inspector:
    let c be the number of dollar bills carried by the player;
    let d be 0;
    if the player carries the wallet:
        now d is the number of dollar bills contained in the wallet;
    let total be d plus c;
    if total is 0:
        say "A hurried inventory [if the player carries the wallet]of your wallet [end if]reveals that you're flat broke. 'I would never resort to offering you a bribe,' you say haughtily. But the inspector is unimpressed. 'In that case,' he says, 'you'll be payin['] the city a hefty fine. That's how the game is played, guy.'";
    otherwise if total is less than 10:
        if d is greater than 0:
            [if the player has any money at all in the wallet, it will be removed and offered to the inspector, who will turn it down:]
            repeat with X running from 1 to d:
                let greenery be a random dollar bill in the wallet;
                now the player carries greenery;
            say "You haul some money out of your wallet and hold it out [run paragraph on]";
        otherwise:
            say "You hold out every dollar you have [run paragraph on]";
        say "to the inpector in a trembling hand, but he only favors you with a disgusted look. 'Not enough,' he says. 'Not near enough.'";
    otherwise:
        [We don't know whether the player has 10 or more dollars in hand, or whether some of the dollars need to be transferred from wallet to hand before being used -- all we know is that between the in-hand dollars and the wallet dollars, there are enough. So we'll start by adding to the in-hand dollars if we need to:]
        if c is less than 10:
            repeat with X running from c to 9:
                [Yes, 9 is correct. If c is 7, for instance, we need 3 more, so we want the loop to run 3 times -- once with X being 7, once with it being 8, and finally with it being 9:]
                let greenery be a random dollar bill in the wallet;
                now the player carries greenery;
            say "You flip open your wallet. [run paragraph on]";
        repeat with X running from 1 to 10:
                let greenery be a random dollar bill carried by the player;
                remove greenery from play;
        say "'Would ten dollars be enough to have you ignore the bare wires, the cracks in the foundation, and the gaps in the pipe joints?' you ask, holding out the cash. 'Oh, sure, heck, whatever,' the inspector replies. In a moment the transaction is concluded. He makes a few careless check marks on his clipboard and moves away, humming a little tune. 'Pennies from Heaven,' it sounds like.";
        remove the inspector from play.
```

Arithmetic

Many computer programming languages are well supplied with slick features for doing advanced mathematical calculations. Inform is not one of them.

This is not a big deal, because there’s no reason why most interactive fiction games ever need to do math. At most, you may need to create a few numbers, add and subtract them, and test their current values. This much Inform can do with ease.

When we need to create and use a number variable, we can do it globally, or we can attach it to an object as a property of that object. It’s pretty much up to you to choose which form you prefer. The first way (making it a global value) leads to easier typing, but programmers who have some experience in object-oriented languages sometimes prefer, as a matter of style, to attach data to objects. Here’s how to create a number using each of those methods:

```
[Global:] Parrot-squawk-count is a number that varies. Parrot-squawk-count is 0.
```

```
[Object property:] The parrot has a number called squawk-count. The squawk-count of the parrot is 0.
```

In each case, the second sentence should be read as including “When play begins:”. We can later change the squawk-count. The syntax for doing this depends on how we’ve defined the variable to begin with:

```
increase parrot-squawk-count by 1; [if it’s a global value]
```

```
increase the squawk-count of the parrot by 1; [if the number is a property of the object]
```

(A note for experienced programmers: All of the properties of an Inform object are public. That is, they’re available to any other code in the game. It’s not possible to make a property private: Inform does not support data hiding. So the distinction between global variables and properties is purely one of personal style. There is no functional difference.)

We can test the current value of a variable like this:

```
if the squawk-count of the parrot is greater than 7:
    now the parrot is dead.
```

The syntax for these operations is outlined on **p. 14.2** of the Documentation, “Numbers.” Other tests that we can use include:

```
if the squawk-count of the parrot is 7;
if the squawk-count of the parrot is at least 7;
if the squawk-count of the parrot is less than 7;
if the squawk-count of the parrot is at most 7;
```

As far as Inform is concerned, all numbers are *integers*. That is, they're whole numbers. The 5Z71 release of Inform provides a way to fake numbers that have decimal places, such as 3.14159, but you can't do a lot with this new tool. For one thing, it only works with named values, not with raw numbers. The Extension called Fixed Point Maths by Michael Callaghan adds a few features that you might find useful — but again, Inform is not really intended for doing calculations.

Warning: If you're compiling to .z5, .z6, or .z8, the largest numbers Inform can handle are a little over 30,000 (which should be plenty for counting parrot-squawks). If you need to use larger numbers, you'll have to compile your game to Glulx.

Doing Calculations

In a role-playing game (RPG), you may want to keep track of various characters' strength and other characteristics. While the game is running, you may need to calculate things like their combat readiness based on various factors. The example below, which closely follows some code Mike Tarbert posted on r.a.i-f, shows how this type of thing might be done. It uses Inform's To Decide syntax to run calculations.

```
A person has a number called strength. A person has a number called luck. A person has a number
called dexterity.

To decide what number is combat-adds of (p - a person):
        let num be a number; [this will default to 0]
        let num be num + the modifier of the strength of p;
        let num be num + the modifier of the luck of p;
        let num be num + the modifier of the dexterity of p;
        decide on num.

[Since the modifier formula is the same for each attribute, we can create a separate routine to
calculate that and reuse it for each one. If the base value (strength, luck, or dexterity) is greater than
12, we'll modify the combat-adds by adding the surplus to combat-adds. If it's less than 9, we'll
subtract the difference from combat-adds:]

To decide what number is modifier of (n - a number):
        let m be a number;
        if n > 12, let m be n - 12;
        if n < 9, let m be n - 9;
        decide on m.

To say stats of (p - a person):
        say "Strength: [strength of p]; Luck: [luck of p]; Dexterity: [dexterity of p].";
        say "Combat Adds: [combat-adds of p]."

[Here's a generic dice-rolling routine:]
To decide what number is a (n - a number) d (m - a number) roll:
        let tot be a number;
        repeat with loop running from 1 to n:
                let x be a random number between 1 and m;
                let tot be tot + x;
        decide on tot.

To char-roll (p - a person):
        change the strength of p to a 3 d 6 roll;
        change the luck of p to a 3 d 6 roll;
        change the dexterity of p to a 3 d 6 roll.

[Purely for testing, the command JUMP will re-roll the player character:]
Instead of jumping:
        say "Re-rolling your character...";
        char-roll the player.

The Lab is a room.
When play begins:
        char-roll the player;
        say stats of the player.

Every turn: say stats of the player.

test me with "jump / g / g / g".
```

Several things about this example are worth study. In the “Instead of jumping” rule, for instance, we have “char-roll the player.” This calls the block of code just before it, “To char-roll (p – a person)”. This is Inform’s standard way of creating new functions. Once the “To char-roll” code is created, we could char-roll any character in the game. But the compiler wouldn’t let us char-roll anything other than a person.

In the odd-looking syntax, “change the strength of p to a 3 d 6 roll”, the dice-rolling routine rolls 3 dice that are 6-sided, adding the results to give a value somewhere between 3 and 18.

Testing Conditions

We’ve seen many examples in this book that test conditions in the model world. The keyword in each case is “if”. **Page 11.5** of the Documentation (“If”) explains how to create and test conditions. The point of testing a condition is for the game to make a decision about what to do next. If the condition is true, we want the game to do one thing; if the condition isn’t true, we want something different to happen. Here’s a simple example:

```
if the player carries the big stick:
	end the game in victory;
otherwise:
	end the game in death.
```

Inform is a bit unusual in that it also allows us to test a condition using “unless”. Unless means the opposite of if:

```
unless the player carries the big stick:
	end the game in death;
otherwise:
	end the game in victory.
```

The condition being tested in those examples was “carries”, but we can test almost any condition — is, wears, and so on. We can test whether a truth state is true or false, or whether a property of an object has a certain value. For instance, if we’ve written that temperature is a kind of value, and that the temperatures are frigid, tepid, lukewarm, and boiling hot, and also told Inform that the sea has a temperature, then we could test the temperature of the sea like this:

```
if the sea is boiling hot:
	say "Ouch! I’m getting blisters!"
```

Two or more if-tests can be strung together in one line, like this:

```
if the cat is on the couch and the catnip is on the couch and the dog is not on the couch:
	say "The cat goes a little crazy."
```

But in this type of construction, Inform insists that each phrase in the if-test be spelled out in full. The condition shown below makes perfect sense to a human reader, but the syntax is too complicated for Inform to understand:

```
if the cat and the catnip are on the couch: [Error!]
```

Creating New Phrases

Sometimes you may need to have your game do several things at once. A good way to take care of this is to write what other programming languages would call a *function*. This feature of Inform is introduced on **p. 11.13** of the Documentation, “Phrases which use descriptions.” Writing a function in Inform is easy — just use the word “To”, like this:

```
To sound the alarm:
	now the alarm horn is blasting;
	now the burglar is frightened;
	now the guard dog is awake;
	[...and so on...]
```

From anywhere else in your code, you can now sound the alarm, simply by telling Inform that that’s what you want to do:

```
if the burglar carries the jewels:
        sound the alarm.
```

The main reason to write a function of this sort would be if your game may need to sound the alarm from several different places in the code, in response to several different events. Rather than putting the “sound the alarm” code in several places, you can put it in just one place. This way, if you need to edit it later, you only need to make the change in one place, which is easier and also reduces the chance of introducing a bug into your game.

If you need to, you can write a function that can be applied to any number of different objects in your game. Here’s a slightly artificial example that shows the syntax:

```
To blast open (box - a container):
        now the box is open;
        now the box is not openable;
        now the box is damaged;
        say “Boom! You blast open [the box].”
```

The word “box” in this example is what computer programmers call an *argument*. This just means it’s a temporary placeholder for some data. In order to run this block of code, you would have to tell Inform what “box” will refer to. This is called “passing an argument to the function.” If your game includes an old trunk and a safe, for instance, you could use the command “blast open the old trunk” or “blast open the safe” in your game. When Inform calls the “To blast open” function in response to a line that reads “blast open the old trunk,” it will know that the old trunk is now the “box” being referred to, so it will operate exactly as if you had written “now the old trunk is open”, “now the old trunk is not openable”, and so on. At the end, the game will report, “Boom! You blast open the old trunk.”

But if you mistakenly try to blast open something that isn’t a container, you’ll have a bug. It’s also important to note that the code above changes the property called “damaged” on the container. If you forget to create the property “damaged” for all of the containers in your game, the code above may cause a run-time error.

Take another look at the syntax in the code above. To write a function that takes an argument (or two arguments, if you need to), you use parentheses, create a temporary name for the argument (“box”), then use a single hyphen, then tell Inform what kind of data you’re planning to send to the function.

Indenting

Computer programmers (including Inform authors) often need to write blocks of code that will be run, line by line, in a certain order. Such blocks often need to branch based on whether a logical test is true or false. The keyword “if” is used in many programming languages, including Inform, to write a statement that will allow the code to branch. In pseudo-code, a short version look be something like this:

if A, do X; otherwise, do Y.

Here, “X” and “Y” would be entire lines of code. A more complex (and more realistic) example might look more like this:

if A, do Q, R, and S; otherwise, do X, Y, and Z.

Again, “Q,” “R,” “S,” and so on would be entire lines of code. In this situation, the compiler needs some way to understand which lines of code to group together into a block. Q, R, and S should be grouped together, and X, Y, and Z should be grouped together — but it would be horribly wrong to have all six grouped together!

Two systems for organizing lines of code into blocks (groups of lines) are in common use. Some languages use indenting. Other languages, such as C, don’t require indenting (though indenting can be used to make their source code easier to read). Languages that don’t use indenting usually put brackets or curly braces (like these {...}) around blocks of code.

The original release of Inform 7 used neither system. Instead, in keeping with the “natural language” idea, the keywords “begin” and “end” were used to set off code blocks. Here is an example (borrowed from Chapter 4 of this *Handbook*) using the syntax format that Inform originally provided:

```
Before taking the apple:
if the player does not carry the apple begin;
if the player carries the basket begin;
if the apple is not in the basket begin;
now the apple is in the basket;
say "You pick up the apple and put it in the basket.";
rule succeeds;
end if;
end if;
end if.
```

This type of syntax can still be used in Inform, as noted briefly on **p. 11.7** of the Documentation, “Begin and end.” And in fact, when you get error messages from the compiler you may still find a reference to “begin;”, even though you didn’t use “begin;”.

As you can see, the internal logic of a block of code that ends with a string of “end if” lines can be hard for a human reader to follow. You can use indentation to make the code easier to read without changing the syntax in any way. Here is the same code with added white space (a big improvement).

```
Before taking the apple:
    if the player does not carry the apple
    begin;
        if the player carries the basket
        begin;
            if the apple is not in the basket
            begin;
            now the apple is in the basket;
```

```
                        say "You pick up the apple and put it in the basket.";
                        rule succeeds;
                        end if;
                end if;
        end if.
```

That white space might be either Tab keys or rows of spaces — it doesn't matter, because the indenting is just to make the code easier for you to read. The compiler ignores it.

But if we're going to use indentation to make the code easier to read, why not dispense with the "begin/end" keywords entirely? After Inform 7's initial release, there was apparently a groundswell of support for the idea of letting Inform authors use indentation *instead* of "begin;" and "end if;". This style of code formatting is sometimes referred to as Pythonesque, because it's used in the popular Python programming language. Below is exactly the same code as in the example above, rewritten to use colons in place of "begin;", and indenting to keep track of where each if block ends. Many Inform programmers today seem to prefer this method:

```
Before taking the apple:
        if the player does not carry the apple:
                if the player carries the basket:
                        if the apple is not in the basket:
                                now the apple is in the basket;
                                say "You pick up the apple and put it in the basket.";
                                rule succeeds.
```

The "end if" statements are no longer needed. I find this much easier to read.

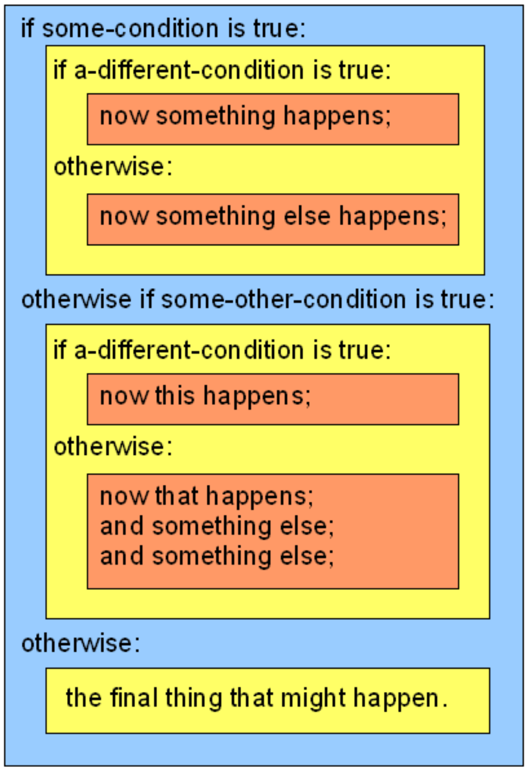
I've found that newcomers quite often don't pay enough attention to how many indents (that is, how many Tab key presses) they need for a particular line of code. If the indenting is wrong, the compiler will get confused. To make matters worse, some email programs turn indents into strings of space characters. They look the same on the screen, but Inform requires Tabs, not spaces. So emailed code can get messed up, even if you've used standard copy-and-paste to paste the code into the email.

On the Macintosh, the source code editor in the Inform program formats paragraphs with "hanging indents." That is, each line of a paragraph after the first one is indented slightly. This is purely to make the code easier to read. The hanging indents have nothing to do with indenting code blocks to create organization, as shown above. They're purely a way of making the code a bit easier to read on the screen.

When do you indent? The rule is simple: After a colon, you indent by one more Tab than the line that ends with the colon.

Basically, a colon means "do the following action(s)." Following every colon is a list of one or more steps that you want Inform to take. We can call this list of steps a *block* of source code. The block of code should be indented by one more level (that is, one more Tab key) than the statement before it — the statement that had a colon at the end. Everything that is indented further than the line that ends with the colon is part of that block of code, so it will run if that line tells it to. When we return to the same level of indentation as the line that ended with the colon, we're done with that inner block of code and ready to proceed with the next statement in the outer block.

Yes, that description is hard to read. The diagram on the next page may help make the organization of code into outer and inner blocks easier to visualize. Each line in each of the colored blocks in that diagram is indented one Tab further than the lines in the block outside of it.



```
if some-condition is true:
    if a-different-condition is true:
        now something happens;
    otherwise:
        now something else happens;
otherwise if some-other-condition is true:
    if a-different-condition is true:
        now this happens;
    otherwise:
        now that happens;
        and something else;
        and something else;
otherwise:
    the final thing that might happen.
```

Here's an example from a recent game of my own that may make this idea a little more real-world. If you don't understand what's going on here, you might want to look back at the section "Testing Conditions." In that section, if statements are explained.

```
Some eyes are part of Elliott. The description is "You can't see your eyes! They're just there."

Instead of closing the eyes:
	if Elliott wears the black blindfold:
		say "Closing your eyes will make no difference. You're wearing a blindfold.";
	otherwise if the location is a crystal room:
		say "You close your eyes for a moment against the dazzle of sunlight on crystal, but a moment later you bump into something and your eyes fly open again.";
	otherwise:
		say "You close your eyes for a moment. Ah, how restful!"
```

This Instead rule will run when the player types CLOSE EYES. The outer block of code in the rule begins with the colon at the end of the first line. There are three lines in the outer block of code. The first line begins "if Elliott wears...", the second begins "otherwise if the location...", and the third is "otherwise:". These lines are indented with one Tab. After "if Elliott wears the black blindfold:", we need to indent by two Tabs to create an inner block of code. This happens to have only one line in it, a say statement — but it could have many more lines.

Note that the statement that begins “say “You close your eyes for a moment...” is only a single line as far as Inform is concerned. It happens to flow down onto the next line of the page, but the fact that the next line isn’t indented doesn’t matter, because there’s no *carriage return* character before it. In the Macintosh IDE, long lines like this have what’s called a “hanging indent,” which makes it easy to see that they’re all part of a single code statement. In the Windows IDE, long lines look the way they do in this book.

Nothing in the Instead rule shown above actually changes the model world; all it does is print out one of three different messages, all of which boil down to “You can try that, but it won’t do any good.” But if the rule included a line like “now the blindfold is transparent;” the organization of the code into outer and inner blocks would be the same.

Too Many Indents

Up through version 5Z71, Inform allows only nine levels of indentation. If you’re writing an extremely complex, embedded set of if-tests and need more than nine levels, you’ll have to reorganize your code in some other way. Fortunately, this is not difficult, and it has the side benefit of making the code easier to read and easier to maintain.

There are several ways to reduce the number of indentation levels. For instance, these two Instead rules are functionally identical, but one uses two indent levels, while the other uses only one:

```
Instead of eating the pretzel:
        if the pretzel is salty:
                say "Recognizing that the salt is the best part, you pause and lick off the salt first.";
                now the pretzel is not salty;
        otherwise:
                continue the action.

Instead of eating a salty pretzel:
        say "Recognizing that the salt is the best part, you pause and lick off the salt first.";
        now the pretzel is not salty.
```

If you need to perform a number of operations (such as changing various values) within a block of code, you can reduce the indentation by using a To statement. Here’s a quick example that expands slightly on a few lines that appeared earlier in the *Handbook:*

```
After attacking the ceramic bowl:
        if the guard is in the location:
                if the guard is awake:
                        sound the alarm;
                otherwise:
                        [more code goes here...]

To sound the alarm:
        if the ceramic bowl is shattered:
```

```
        [more code goes here...]
```

The line “if the ceramic bowl is shattered” has only one indent, because it’s in a separate block of code. If it were embedded after “if the guard is awake” in the “After attacking the guard” rule, it would have three indents.

Headings

Another difference between Inform and traditional programming languages is that, with some important exceptions, Inform doesn’t care where you place rules and assertions. If you want to mix up your source code, leaving related bits strewn out all over everywhere, Inform will let you. If you’re working on the section of the story that has to do with the swords, and you suddenly realize that you need to make sure the men in the tower are wearing chain mail, you can just start a new paragraph and create the chain mail. There’s no need to go find the men in the tower (they could be five hundred lines earlier or later in the file) and put the chain mail in their part of the code.

This is bad programming practice, though. It’s much better to keep related code together. Inform will try (using the Index World tab) to help you find everything in the code, no matter where it is. But if you create your own organization, the writing will go more smoothly, and you’ll end up with fewer bugs.

The natural and normal way to do this is by writing headings and putting all of the code for some specific thing below a single heading. For instance, the chapter on the wizard might include a section called “Conversations with the Wizard.”

The larger your game is, the more important it will become to use headings in this manner. In most programming languages, you can put various parts of the code in separate files on your hard drive. Inform doesn’t allow this, because it’s designed around the idea that what you’re writing is a story or book, which would naturally be one continuous block of text.

Using headings within a single file is almost as convenient as organizing your program in multiple files, and it has some advantages too. After you’ve compiled your game, the Index will display an outline of the source code, listing all of the headings. This makes it easy to move around in a large file — just click on the heading, and the Source pane will jump to it. (If you’ve added code since the last time you compiled, this mechanism may not work perfectly. Click the Go! button and then look at the Contents page of the Index again.)

How you organize the code and give headings to the various sections is almost entirely up to you. As explained on **p. 2.5** of the Documentation, “Headings,” the only requirement is that each heading be on a line by itself, with a blank line before it, and another blank line after it.

Inform recognizes five words as headings: volume, book, part, chapter, and section. When the Index is being constructed, these five headings are considered *hierarchical*, which is just a fancy way of saying that a volume is bigger than a book, a book is bigger than a part, and so on. But you can ignore the hierarchy if you like, and give sections headings in whatever order you like. A well organized outline might look like this:

```
Volume 1 – The Castle
        Book 1 – The Courtyard
                Part 1 – The Courtyard Room
                Part 2 – Scenery in the Courtyard
                Part 3 – The Guard
                        Chapter 1 – The Guard Himself
                        Chapter 2 – Conversations with the Guard
                        Chapter 3 – Being Chased by the Guard
        Book 2 – The Dining Hall
```

...and so on.

Many Inform programmers prefer to number their volumes, books, parts, and so on, as shown above. But the numbers don't have to be in order, and numbering is not even required. All that's required is that there be *something* after the heading word. If you just type Chapter and then forget to put anything after it, the compiler will complain. (Also, you can't put a colon immediately after a heading word.)

Giving each heading a name is a very good idea. This will help you understand what you're seeing in the Index.

One of my students put a bunch of related material in a single chapter and then asked, "How can I tell Inform, 'Chapter 3 starts now'?" This is a very reasonable question, but the answer is — you can't. The headings are strictly a way to organize your source code. They have no effect whatever on what happens when the game is being played. To switch to a new set of circumstances at a certain point in the game, you need to use scenes, as explained in Chapter 8 of this *Handbook* and **Chapter 10** of the Documentation, "Scenes."

Loops

Sometimes a computer program needs to perform a certain operation over and over. If it needs to do the same operation 50 times, there's no sense in writing out the same block of code 50 times. Instead, we write it out once and then execute it over and over until we're done. The process of doing this is called a loop.

Loops are used less in interactive fiction than in many other types of programming, but they definitely have their uses. We've already seen loops a few times in the *Handbook,* for instance on p. 228, where the loop was used to move a bunch of dollar bills. Here's a more straightforward example.

Let's suppose you have a set of rooms in a region called the Underground Area, and let's suppose the player has just thrown a circuit breaker that plunges the entire Underground Area into darkness. In that case, you could write a routine in which you manually list each room in the Underground Area and make it not lit — but it would be easier, and less likely to introduce errors, if you write a routine that automatically loops through all of the rooms in the Underground Area and makes them dark, like this:

```
After switching off the circuit breaker:
	repeat with R running through rooms in the Underground Area:
		now R is not lit;
	continue the action.

After switching on the circuit breaker:
	repeat with R running through rooms in the Underground Area:
		now R is lit;
	continue the action.
```

In these two After rules, we’re using the temporary variable R to refer to a room. (We could just as easily have called it Abercrombie — the use of R to mean “room” is not special.) The loop is created by the word “repeat”. To execute the loop, Inform first makes a list of the rooms in the Underground Area and then runs the code for the loop (which in this case consists of the single line “now R is not lit”) a number of time. Each time Inform goes through the loop, the variable R refers to a different room. So the loop has the effect of turning off the lights in each room in the Underground Area, one by one. When it gets to the end of the list of rooms in the Underground Area, the loop stops.

The basic syntax for how to use loops is on **p. 11.6** (“While”), **p. 11.9** (“Repeat”), and **p. 11.10** (“Repeat running through”) of the Documentation.

Where to Learn More

If you’re having trouble finding the right phrase with which to do something, you might try spending an hour or two studying the Inform 7 Syntax document. This is available for download at http://inform7.com/learn/documents/I7_syntax.txt. It includes almost no explanations, but the syntax that’s needed for almost any programming task in Inform is to be found in it. Because it’s a plain text document, you can search it after opening it in a text editor or word processor.

Chapter 10: Advanced Topics

To round out *The Inform 7 Handbook,* we'll take a quick look at some of the advanced topics that Inform authors sometimes need or want to know about. We're not going to provide every detail about any of these topics — that would take a whole other book. But if you're wondering what may lie ahead in your adventures with Inform, we'll try to get you started on the right foot. For more details about how to use the features described here, you'll need to read the Documentation and perhaps post messages to the newsgroup rec.arts.int-fiction requesting help.

Tables

Tables are Inform's way of organizing complicated blocks of data. **Chapter 15** of the Documentation, "Tables," will tell you a great deal about tables, but it won't, at first glance, give you a very clear idea why you might want to use a table. I'm pretty sure you can write a complete and satisfying game without ever using tables. They're a specialized tool, used mainly when you have long lists of stuff that you want to be able to get at during the game. Tables are used in creating hint menus and conversation menus, for instance.

When I asked the Inform experts (on the newsgroup rec.arts.int-fiction) for ideas about why an author might want to use a table, Emily Short posted an excellent list. I'll include her suggestions here, with only minor editing, in case they might give you an idea or two for your next game.

Emily Short on Using Tables

The most common uses for me are these:

— To manage background-event text. This might take the form of events that need to happen in sequence, during a scene (see "Day One"), or it might be a set of randomized atmosphere texts to be printed under certain conditions. Mikael Segercrantz's Atmospheric Effects Extension is an especially thorough expression of this concept.

I use this trick *all the time*.

— To store the mental equivalent of inventory. In my WIP "Alabaster," for instance, there is a long, long list of facts the player might know, each of which has a short summary text to be printed by THINK if the player has in fact discovered this fact. This information is stored like this:

```
Table of All Known Facts
fact              summary
snowshoes-worn    "She is wearing snow shoes."
apple-pie         "She really likes apple pie."
```

...and so on, for dozens of lines. (Actual facts changed to protect against spoilers.)

— To give an NPC pert replies to being asked to do dumb things. I find it most convenient to make a table of the different rules that cause an action to fail, and then attach some reply text to each one. ("Generation X" demonstrates this.) I could also handle this with a series of individual “unsuccessful attempt” rules, but because there would have to be a large number of these, I find the table is easier to take in at a glance.

— To construct any kind of consultable object in which the player needs to look things up: books, computers on which you conduct Google-style searches, or the sort of NPC that exists chiefly to answer questions. (Most of mine don’t, which is why I usually don’t use tables to construct them; but I could imagine a robot librarian that would be best implemented by a table.) Several reviewers commented favorably on the computer database search in “Floatpoint,” and this would have been vastly harder to set up without tables with topic columns.

— To store flexible schedules. Inform has various time functions that make it possible to write an inflexible schedule (X happens at a given time of day, no matter what), but sometimes I like to have a free-floating schedule that could start at any time. My Extension “Transit System” is a good example of this in action: Each row contains a number of minutes and the name of a room, which is the stop where the bus (say) ought to arrive after the requisite delay.

There are also some obscure programmatic uses to them, of which the most obvious is perhaps:

— As an aid in Extensions, whenever I want to create a sequence of things to which the author of the game may need to add. Because tables can very easily be amended or added to, they make good structures for an Extension designer to use. “Complex Listing” exemplifies this: It offer several kinds of list-building options but allows the game author to add almost any kind of addition to the selection.

In practice, the things you can do with a table are not *that* numerous. They basically boil down to:

To find your place in a table, you can step through the whole table row by row. You can choose a specific row, by row number or by looking for a specific value in one of the columns, or by selecting a blank row with nothing in it yet.

Manipulating the contents of a particular row, once chosen: You can read and use a table entry for something (for instance, to print text or set a local variable). You can write something new in a table entry. Or you can blank out a row of entries, eliminating them permanently

You can manipulate the whole table by sorting the table to put one of the columns in a specific order.

Even topic tables are just a special case of this, in that they have a column with a special name (“topic”) and they allow the author some shorthand for looking up rows in the table based on the player’s input — but this is still just row-choosing stuff.

In a nutshell, here’s what you need to know about tables in order to start using them:

The data in tables is organized into rows and columns. Each column has a name, which is how you’ll refer to it while writing your game. The table itself has a name too. The rows don’t have names; they’re numbered.

Each column can contain only one type of data. The first column might contain things, the second column numbers, and the third and fourth columns text. If you try to mix up the columns, for instance putting a number where text belongs, Inform won’t let you.

When you create a table, you separate the data within each row by using one or more Tab characters. Rows of space characters look the same to you and me, but they won’t do the job. At the end of each column, you hit a single Return/Enter, and then start the next column. Tables with long texts tend to look quite jumbled on the Source page, but if you follow these rules and ignore how the table looks on the screen, it will work the way you want it to.

Let’s create a table, so that we can refer to it in the rest of this section:

```
A furry menace is a kind of thing.

The gopher is a furry menace in the Meadow.
The muskrat is a furry menace in the Meadow.
The chipmunk is a furry menace in the Meadow.
The mole is a furry menace in the Meadow.

Table of Obnoxious Mammals
critter         name            weight
gopher          "Herman"        17
muskrat         "Vivian"        12
chipmunk        "Abercrombie"   9
mole            "Edith"         27
```

The first row under the table name contains the column headings (critter, name, weight). These are just abstract words, as far as Inform is concerned. I’d recommend not using a word that Inform is using for something else, or that you’re using elsewhere in your own code. You should avoid column headings like “location” and “container”, for instance.

To use the data in any cell in the table, we can refer to it by the line number and column heading. For instance, “the name in row 4 of the Table of Obnoxious Mammals” is the text “Edith”. If you’re familiar with computer programming, you’ll understand when I say that Inform’s table rows are 1-indexed, not 0-indexed. In plain English, the number of the first row is 1, not 0.

You can also get at the data in the table by cross-referencing the data you want with the data in some other column, like this:

```
say "The name of the muskrat in the Table of Obnoxious Mammals is [name corresponding to the
critter of muskrat in the Table of Obnoxious Mammals]."
```

Note the slightly weird syntax here. We have to say “corresponding to the critter of muskrat” — we can’t say, “corresponding to the muskrat critter.”

You may sometimes need to change the data in a table during the course of the game. Inform’s standard “change … to” or “now … is” syntax will do the job:

```
now the weight in row 1 of the Table of Obnoxious Mammals is 8;
```

It’s often useful to look at all of the data in a table, one row at a time. Here’s one way to do that:

```
repeat with N running from 1 to the number of rows in the Table of Obnoxious Mammals:
	if the weight in row N of the Table of Obnoxious Mammals is greater than 15:
		say "[Name in row N of the Table of Obnoxious Mammals] is rather obese."
```

The phrase “repeat with N running from 1 to the number of Rows in the Table of Mammals” has several elements. The phrase “repeat with” creates a *loop*. Inform will run through the code block below the “repeat with” statement over and over. The first time through the loop, N will be 1. The second time through the loop, N will be 2. And so forth. N is a *loop counter;* the words “running from 1 to” are what tells Inform that N is a number, and that it’s a loop counter. The phrase “the number of rows in the Table of Mammals” should be obvious; in this case it’s 4, because our silly little table has four rows. So the loop will run four times, and then stop.

While the loop is running, we can use the number N (which is different each time Inform goes through the loop) to do various things to the data in the table. Mainly, we can look up data in row N, or we can change it.

Here’s a more concise way of doing the same thing, without using a loop counter:

```
repeat through the Table of Obnoxious Mammals:
	if the weight entry is greater than 15:
		say "[name entry] is rather obese."
```

As mentioned on **p. 15.12** of the Documentation, “Listed in...,” when Inform looks at some data in a table row, that row is automatically chosen. This makes it easy to refer to the other data in the row.

Because we defined the objects that show up in the table as of the furry menace kind, we can write a rule that will apply to any furry menace:

```
Instead of examining a furry menace (called FM):
	if FM is a critter listed in the Table of Obnoxious Mammals:
		say "[The FM] is named [name entry]."
```

Here, we don't need to say "[name corresponding to the FM in the Table of Obnoxious Mammals]" — in fact, we can't say it, because Inform won't know what it means. Because the row has already been chosen automatically, "[name entry]" will work fine.

Sometimes you may need to change the data in a table entry. You can do this by referring to an entry in another column of that same row, which is convenient, because it means you don't need to know the number of the row the data is in. Continuing with our rather silly example, we're going to create a new action, altering, which will do nothing except change the entry in the name column:

```
Altering is an action applying to one thing. Understand "alter [furry menace]" as altering.

Carry out altering:
	now the name corresponding to a critter of the noun in the Table of Obnoxious Mammals is
"Judy Garland";
	say "Name altered."
```

The point of this example is to show the syntax. Two of the columns are called name and critter, so "now the name corresponding to a critter of the noun..." will let us alter the name of any of the critters to "Judy Garland" by typing ALTER MOLE in the game, or ALTER MUSKRAT.

The Skein

As explained in **pages 1.7 and 1.8** of the Documentation ("The Skein" and "A short Skein tutorial"), the Skein and Transcript panels are used to replay series of commands and examine the output. All the time you're testing your game in the Game panel, Inform is quietly recording all of your commands and all of the output from the game. The commands are added to the Skein.

At any time you can switch to the Skein and double-click on a node (the Documentation calls them "knots"), and the game will first be compiled and then replayed from the start up to that knot. The Skein is like the Replay button, except that it can replay *any* playing session, not just the most recent one.

As you work on your game, the Skein can get pretty crowded with branches (Inform calls them "threads") that are no longer needed. You can trim these out by right-clicking (Mac: Ctrl-clicking) and choosing "Delete all below". But you won't be allowed to delete the currently active thread — the one that represents the game that's currently in progress in the Game panel. If you want to do that, you first need to click the Stop button in the toolbar. If you want to delete everything in the Skein, you can use the Trim button. This will get rid of everything except the currently active thread.

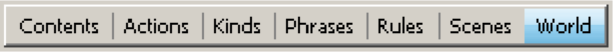
As your game gets closer to being finished, the Skein will become quite useful. You can play the entire game manually from start to finish — until you get to the ***** You have won ***** message — and then open the Skein and *lock* the thread you've just created. Locking a thread will stop Inform from deleting it, even if you use the Trim button. In the Transcript panel, you can *bless* the output transcript for this thread. Blessing tells Inform that this is the output you hope to see in the final, released version of the game. Blessing has no actual effect on the game itself; it's just a record-keeping function. It stores a transcript so that you can look at it later.

After making further changes in your game (to eliminate bugs, for instance, or to add features suggested by your testers), you can run the game again using the locked thread, as a quick check to make sure you haven't done anything that makes the game unwinnable. You can then inspect the output in the Transcript to see what changes have appeared.

There's more to using the Skein and Transcript than we've covered in this brief overview. After you've been writing in Inform for a while, you'll start to see how useful these features can be.

Using the Index

I really shouldn't have buried a discussion of the Index panel in the Inform IDE way back here in Chapter 9, but I didn't know where else to put it. Until you've learned a bit about Inform programming, what you see in the Index may not make a lot of sense. Once you've started working with Inform, you'll find it extremely useful.



Across the top of the Index panel are seven buttons. By clicking them, you can view Contents, Actions, Kinds, Phrases, Rules, Scenes, or the World. The main point of the Index, as you might expect, is to make it easy to find things in your project.

Many of the items listed in the Index have clickable buttons beside them. Clicking an orange button with a curving arrow will take you to the place in the Source where that item is first mentioned. Clicking on a gray magnifying glass button will open up another page within the Index, where you'll see more detail. Clicking a blue question-mark button will take you to the page in the Documentation where the item is discussed.

Contents

If you've used any Headings in your game (see Chapter 8), they'll be listed at the top of the Contents page. This list is similar to the list in the Contents page of the Source panel. Below the list of headings (titled "Contents") are two more lists — the named values in the game and the tables used. If you've created a named value, like this:

```
Beauty is a kind of value. The beauties are stunning, cute, okay, geeky, creepy, and disgusting.
```

...then you'll see your new value near the bottom of the list of named values, below Inform's built-in named values and those in any Extensions you're using.

Actions

The page starts with a list of the actions that are built into Inform, followed by a list of the actions defined in Extensions and then the actions you've defined in your own code. If you click on a magnifying glass next to a built-in action, you'll find a list of the rules that may affect this action.

What's handy is that, while compiling the game, Inform generates a list of anything in your source code that relates to a built-in action. In one of my games, for instance, I wrote several Before and Instead rules about touching specific objects in the game. When I click on the magnifying glass next to the touching action, I see a list of all of these rules — "Before touching something that is worn by Zarbolphung", for example.

Below the list of actions is an alphabetical list of the commands available to the player. If you want to know what action is triggered when the player uses the command CUT, for instance, this page gives you a quick way to look it up.

Kinds

A "kind", in Inform, is a lot like what other programming languages call "classes." A kind is not an object in your game, in other words — it's an abstract template that you can use to define new objects quickly. Inform's built-in kinds include things like room, door, container, supporter, and backdrop. You'll find these in the Kinds Index, along with any new kinds you've created for your game. You'll also find a complete list of the objects in your game, grouped by the kind of object they are.

Phrases

The Phrases page (or Phrasebook page, in the Macintosh IDE) is, among other things, a quick reference manual in which you can look up the way Inform expects your code to be phrased.

If you've used any To Say phrases in your game, you'll find them listed here. You'll also find any named values that you've used in your code.

Rules

This page of the Index has been extensively revised for release 5Z71. The rules you've written for your game will appear here, as will the rules in the Standard Rules and any rules defined in the Extensions that your game includes.

Scenes

If you've added scenes to your game (see p. 202), the Scenes Index will list them. It will also show how they start and how (or whether) they end. By clicking the orange arrow next to a scene listing, you can jump directly to the code that creates the scene.

World

The World tab provides lists of all of the things in your model world. One list is organized by room, and another is alphabetical. Inform will also attempt to draw a map showing your rooms and the connections between them.

As you add more rooms, unfortunately, this map will get to be a mess. It's possible to clean it up using the commands shown on **p. 22.17** ("Improving the index map") of the Documentation, but since it's strictly for your own use while writing the game, putting a lot of extra time into tweaking it may not be worthwhile.

Replacing Rules in the Standard Library

If you open up the Extension called the Standard Rules, you'll find the guts of Inform 7. Or at least, the higher-level guts; there's a lower level, as explained at the end of this Chapter, in the section "What does Inform 6 Have to Do with Inform 7?" The Standard Rules are not exactly light reading, but you can start to get a better idea of what's going on in Inform by skimming them. You'll find, for instance, this bit of code:

```
Check an actor taking (this is the can't take scenery rule):
	if the noun is scenery, stop the action with library message taking
		action number 10 for the noun.
```

This is the rule that produces the output "That's hardly portable" if the player tries to take something that's scenery. If you only want to change the text of the message, you can do so using the very comprehensive Extension called Custom Library Messages by David Fisher. And of course you can intercept the attempt to take any *specific* item of scenery using an Instead rule. So it's not too likely you'll need to tinker with this particular rule. But if you should need to do so, you can.

The way to do it is *not* to edit the Standard Rules file. You can edit the file if you feel compelled to, but you're inviting disaster. Inform provides a more elegant solution. The rule above, like most of the rules in the Standard Library, has a name. It's called the can't take scenery rule. You can easily replace this rule, using the following simple syntax:

```
Check an actor taking (this is the new can't take scenery rule):
	if the noun is scenery:
		say "[one of]Oops -- better not![or]No, way, Jose![or]I can't let you do that, Dave.[at random]";
		stop the action.

The new can't take scenery rule is listed instead of the can't take scenery rule in the check taking rulebook.
```

That last sentence causes your new rule to replace the one in the library.

Indexed Text

We've already met indexed text briefly, in the chapter of the *Handbook* on characters. Indexed text is a kind of data in Inform. It's like ordinary text, but more flexible. For example, we can search indexed text and replace words in it with other words while the game is running. This topic is explained in considerable detail in **Chapter 19** of the Documentation, "Advanced Text." The most common reason to use indexed text in a game is probably to alter the player's input before processing it. This is done with an After Reading a Command rule, like this:

```
After reading a command:
        let T be indexed text;
        let T be the player's command;
        replace the regular expression "tell jack to" in T with "jack,";
        replace the regular expression "ask jack to" in T with "jack,";
        change the text of the player's command to T.
```

If you follow this syntax precisely, you should be able to do most of what you need to do with the player's commands. In the rule above, T is a temporary, local variable. First we create it, then we fill it with data (the player's command). After altering the data in T in whatever way we may need to, we copy the data in T back to the player's command. That's what happens in the last line.

Helping the Parser

Sometimes a word in your game will be ambiguous — that is, the parser will be able to interpret it in two or three different ways. When possible, you should use unique words to refer to every object in your game, but sometimes that's just not possible.

A handy trick for helping the parser deal with this type of ambiguity is to write a Does The Player Mean rule. The way to do this is explained on **p. 16.18** of the Documentation, "Does the player mean...." Other useful techniques are shown on **pages 16.17** ("Context: understanding when"), **16.19** ("Understanding mistakes"), and **16.20** ("Precedence"). One of my favorite techniques (from 16.17) is writing Understand When rules. We've seen a couple of these elsewhere in the *Handbook*. Such a rule might look like this:

```
Understand "cracked" and "chipped" as the Ming vase when the Ming vase is broken.
```

When you put this rule in your code, the words "cracked" and "chipped" will not be understood as referring to the vase if the vase is not broken. Writing conditions that refer to specific scenes is also useful:

```
Understand "dismantled" as the robot when Emergency Repairs is happening.
```

Pronouns

Inform's parser tries to make intelligent guesses about what the player means when she uses words like "it", "them", and "him" in commands. But occasionally this mechanism will go astray. In a few cases, such a lapse can seriously confuse the player. Consider this transcript:

```
>put bullet in revolver
You put the bullet into the revolver.

>point it at dave
Nothing happens.

>point revolver at dave
Dave backs away from you, saying, "No! No!"
```

In this case, the player understands that "it" is the revolver, but the parser thinks "it" is the bullet. If the player only uses the command containing "it", she will conclude (erroneously) that Dave is not intimidated by the loaded revolver.

The Plurality Extension by Emily Short has a nice widget that can solve this particular problem. After including Plurality, you can use the phrase "have the parser notice" in your code:

```
After inserting the bullet into the revolver:
	have the parser notice the revolver;
	continue the action.
```

To handle more complex cases, Ron Newcomb has written a handy Extension called Pronouns.

Clearing the Screen

Some authors like to clear the screen when a new section of the story begins. This can be a nice effect if the player is teleported to an entirely new time or place, for instance. Here's how to get that effect:

```
Include Basic Screen Effects by Emily Short.

After pushing the big red button:
	clear the screen;
	say "[line break]";
	say "You feel a brief tingling sensation....";
	move the player to Phobos;
	if the player does not wear the helmet:
		say "A little too late, you realize you've forgotten to don your helmet. There's no air on
the surface of Phobos....";
		end the game in death.
```

The key command here is obviously, "clear the screen"; the rest is just window dressing. This command is defined in Basic Screen Effects.

The reason for putting in a line break after clearing the screen is because some IF interpreters print the first line after a clear-the-screen command behind the status bar (at the top of the window), which will make it invisible. The line break will cause “You feel a brief tingling sensation....” to appear reliably on all interpreters after the screen has been cleared. Note, however, that if your game creates a status bar with two lines of text, you’ll need to use two line breaks.

Boxed Quotations

Some authors like putting a boxed quotation at the beginning of the game, or even at the beginning of each new section of the story. Inform supports this effect, but only in the most basic way. The first issue you need to be aware of is that text substitutions such as "[italic type]" are ignored. (This is explained on **page 5.13** of the Documentation, “Displaying quotations.”) As a result, double quotation marks can’t appear in boxed quotations at all.

The second issue is that boxed quotations created with the standard coding mechanism won’t be displayed in games that are compiled to Glulx. If you’re compiling to Glulx, you’ll need to use an Extension, such as Glulx Boxed Quotation by Eliuk Blau. This Extension works, but by default it puts an ASCII border around the quotation. While ugly, this border is intended to allow a box to appear around the quotation in interpreters that don’t support reversed coloring (white type on a black background). Because Inform doesn’t provide a mechanism with which your game can determine whether this feature is implemented in the interpreter a given player is using, you can get inferior results either by including the ASCII border, or by not including it.

The third issue is that, for some bizarre reason, a boxed quotation can be displayed only once. You can dodge around this limitation with the Extension called Repeat Boxes by Dave Robinson, but it doesn’t work well. It splots the quotation at the upper end of the interpreter’s text window, directly on top of whatever already happens to be there.

The fourth issue is that the Documentation doesn’t mention the Extension called Basic Screen Effects by Emily Short. This contains a couple of useful commands, including one that makes it easy for the game to pause after the quotation is displayed and wait for the player to hit the spacebar.

Here’s a stub of a game that implements a boxed quotation with a pause after it:

```
Include Basic Screen Effects by Emily Short.

The Bartlett's Familiar is a room.

When play begins:
        display the boxed quotation
        "'I'm thinking, I'm thinking!'
        --Jack Benny";
        show the current quotation;
        pause the game.
```

This will work decently to display a quotation at the start of the game, but if you need to do anything more sophisticated with boxed quotations, you’ll probably need to switch from Inform to some other programming language. It’s possible that this situation will be improved in a future version of Inform, but no guarantees.

I6 Constants

As your game is compiled, Inform goes through two processes. First, your source text and all of the extensions you're using (plus the Standard Rules) are turned into Inform 6 (I6) code. The I6 compiler then runs, producing a playable .z5, .z8, or glulx game. Occasionally, the compilation process will stop at the second stage. This can happen for several reasons, but one of the more common and less troublesome reasons is because you're using too much of something, and I6 thinks it's running out of room.

The Standard Rules define a series of constants, each of which controls the amount of memory available in the game for a certain type of data. Here's the list:

```
Use MAX_ARRAYS of 1500.
Use MAX_CLASSES of 200.
Use MAX_VERBS of 255.
Use MAX_LABELS of 10000.
Use MAX_ZCODE_SIZE of 50000.
Use MAX_STATIC_DATA of 120000.
Use MAX_PROP_TABLE_SIZE of 200000.
Use MAX_INDIV_PROP_TABLE_SIZE of 20000.
Use MAX_STACK_SIZE of 65536.
Use MAX_SYMBOLS of 20000.
Use MAX_EXPRESSION_NODES of 256.
```

If you receive a message from the compiler saying that one of these values has been exceeded, all you need to do is write a line at the top of your source code increasing it. (It goes without saying that you should *never* edit the Standard Rules themselves.) For instance:

```
Use MAX_STATIC_DATA of 360000.
```

With a large game, I've had to increase MAX_DICT_ENTRIES to 5000 to make room for more vocabulary words. (MAX_DICT_ENTRIES doesn't seem to be in the Standard Rules. I have no idea where its default value is defined.)

Increasing a value is easy to do — but you might be wondering, what's going on here? According to Inform guru Andrew Plotkin, in a post on rec.arts.int-fiction, MAX_STATIC_DATA "is the compiler's workspace for array data. I7 uses this for tables, and for storage space for relations, indexed text, dynamic lists, and other on-the-fly work. The short answer is no, [needing to increase it] shouldn't alarm you; 360 kilobytes is pocket change for computers these days. On the other hand, a future Glulx interpreter running on a phone or a Web browser might not be so sanguine. If you're doing something which costs a huge amount of memory (such as a many-to-many relation), and it's not necessary [to your game], getting rid of it can only help performance."

Relations

Relations are explained in **Chapter 13** of the Documentation. They give Inform some extra power, but seeing how best to use that power may not be easy. Relations can do at least two kinds of things, and probably others — I haven't used them much, so I'm not sure.

First, relations can allow us to write about what's going on in the model world in ways that are shorter and easier to understand when reading the code. Second, relations can be used to manage how bunches of objects … well, how they relate to one another.

For a couple of examples of the first usage, see **p. 13.9** of the Documentation, "Defining new assertion verbs." There, some new verbs are defined, which can then be used in code. Here's another example, this one my own:

```
Proximity relates a thing (called X) to a thing (called Y) when the holder of X is the holder of Y. The verb to jostle (he jostles, he jostled, it is jostled, he is jostling) implies the proximity relation.
```

If you need to make a decision in the game based on whether two objects are in the same container, or on the same supporter, or in the same room (directly in the room — on the floor), or carried by the same person, you would normally write something like this:

```
if the holder of the apple is the holder of the orange:
```

After creating a proximity relation as shown above, though, you can simplify this test a bit by writing:

```
if the apple jostles the orange:
```

I've defined the word "jostles" using a relation. Any unused word would serve — "snuggles" or "elbows", for instance (but probably not "is able to touch", since Inform already knows how to test whether a character "can touch" an object).

Various Examples in Chapter 13 of the Documentation show how the interactions among groups of people can be managed using relations. These Examples are worth studying in detail. If you're wondering whether it would be worth your while to spend the time on it, here's an imaginary example that may make the use of relations slightly easier to understand.

Imagine a game in which the player plays the part of Cupid, complete with bow and arrow. To win the game, you need to shoot Jason with an arrow, shoot Jennifer with an arrow, and then get Jason and Jennifer into the same room so that they fall in love.

If they wander into the same room, the software will need to be able to figure out whether each of them has been shot with an arrow and is ready for love. That's the easy part. All we need to do is write a few assertions along the lines of "A person can be ready-for-love." If Jason has been shot and is ready-for-love, he'll fall in love with Jennifer when he sees her (assuming we write code that will make him do so). But Jennifer might not have been shot yet, so she might not fall in love with Jason when she sees him.

There’s no real need to use relations to manage this game. You can do something simple, with properties, like this:

```
Jason can be ready-for-love or not ready-for-love. Jason can be loving-Jennifer or not loving-Jennifer.
```

...and similar properties for Jennifer. When Jason is loving-Jennifer and Jennifer is loving-Jason, the player has won the game. We don’t need relations here, because these two properties will do the job nicely.

But now imagine that the game includes Steve, Bill, Ted, Ralph, and Jason; and also Jennifer, Susan, Beth, Amy, and Helen. Any of the men can fall in love with any of the women, and vice-versa! (In the interest of not offending our more traditionally minded readers, we’ll ignore the other possibilities.) If we try to use properties, the list of properties each character will need is going to be rather long, and managing these properties in such a way as to avoid bugs will be quite tricky. To manage the potential romantic tangles that can arise during the game in a more reliable way, a better approach will be to use relations.

The code for this game would almost certainly include the example sentence on **p. 13.5**, “Loving relates various people to one person.” We would have to write more code in order to cause characters who have been shot with one of Cupid’s arrows to fall in love with the next suitable person they see. I’m not going to do that here, but it’s not actually a bad idea for a game. I hope someone will try it. Once a character is ready-for-love and sees an appropriate romantic object, all we need to write is:

```
now Steve loves Susan;
```

If Steve has previously been in love with Amy, that one line of code will automatically make him fall out of love with Amy, because the statement “Loving relates various people to one person” allows a person to love only one other person at a time. That’s the power of relations: Relations can manage the various combinations a lot more easily.

If you’re using relations in your game, you can take advantage of the debugging command RELATIONS, which prints out a list of all the relations that are currently active among the objects in your game.

Here’s a more complete (and more practical) example of how to use relations. In this short game, we want the player to have to stand on a chair in order to touch the chandelier. But it’s not enough to stand on the chair: The chair has to be positioned correctly beneath the chandelier. In addition to a new action (putting it beneath), we need to define a relation that will keep track of whether something is beneath something else. Whenever the player picks up an object, we need to get rid of the relation. And just to keep things tidy, we’ll also create a new property, high or low. The player will only be able to put things beneath things that are high. In a game where the player is allowed to put a coaster underneath a glass, or a silver dollar underneath a sofa cushion, the high/low property might get in the way, and the rules for the putting it beneath action would naturally be more complicated. But this example could easily be adapted so as to force the player to put a chair beneath a high shelf in order to get something off of the shelf, for example.

```
A thing can be high or low. A thing is usually low.

The Living Room is a room. "A large, old-fashioned room lit by a crystal chandelier."

The crystal chandelier is scenery in the Living Room. The description is "It shimmers with light." The
chandelier is high.

The chair is an enterable supporter in the Living Room. It is not fixed in place. The description is "A
sturdy chair[if the chair is near the chandelier]. It's positioned directly beneath the chandelier[end if]."

Proximity relates things to each other. The verb to be near implies the proximity relation.

Putting it beneath is an action applying to two things and requiring light.

Understand "put [something] underneath/beneath/under [something]", "push [something]
underneath/beneath/under [something]", "place [something] underneath/beneath/under [something]",
"set [something] underneath/beneath/under [something]", and "position [something]
underneath/beneath/under [something]" as putting it beneath.

Check putting it beneath:
        if the second noun is not high:
                say "[The second noun] lack[if the second noun is not plural-named]s[end if] the
requisite elevation." instead.

Carry out putting it beneath:
        now the noun is in the location;
        now the noun is near the second noun.

Report putting it beneath:
        say "You place [the noun] on the floor directly beneath [the second noun]."

Instead of putting the chair beneath the chandelier:
        if the chair is near the chandelier:
                say "The chair is already beneath the chandelier.";
        otherwise if the player is on the chair:
                say "You'll have to get off of the chair if you want to do that.";
        otherwise:
                if the player carries the chair:
                        move the chair to the location;
                now the chair is near the chandelier;
                say "You place the chair beneath the chandelier."

After taking something:
        if the noun is near a thing:
                now the noun is near nothing;
        continue the action.

Check touching the chandelier:
        if the player is not on the chair:
                say "You can't reach it." instead;
        otherwise if the chair is not near the chandelier:
```

```
		say "It's out of reach. Maybe if you put the chair underneath the chandelier first, you could reach the chandelier." instead.

Report touching the chandelier:
	say "The crystal is cool to the touch, and a little dusty. The entire chandelier sways gently as your fingers brush it, and makes a soft, musical tinkling sound.";
	rule succeeds.

The player carries a bowling ball. [For testing purposes.]

Test me with "touch chandelier / stand on chair / touch chandelier / get off / put chair under chandelier / stand on chair / touch chandelier / put ball under chandelier / relations".
```

Adding Hints

My personal belief (some people don't agree) is that almost any work of IF that includes puzzles should have a complete set of built-in hints. The bad thing about hints is that they can make it too easy for the player. Instead of exercising her brain-power, she can just look up a hint, slam through the puzzle, and move on to the next part of the game.

On the other hand, a player who gets stuck in the middle of a game and has no way of getting unstuck may just set the game aside and never come back to it. If you care about having players enjoy your game from start to finish, I hope you'll at least consider providing a hint menu. It's a little extra work, but many players will appreciate it.

A well-written series of hints will first nudge the player gently toward the solution of each puzzle. If the player asks for more hints, they'll become broader and more obvious. The last hint in each list should provide a complete, step-by-step solution to the puzzle.

A good Extension to use for adding hints to your game is Adaptive Hints by Eric Eve. The documentation for this Extension is thorough and easy to follow, so there's no reason to discuss it here. One nice thing about Adaptive Hints is that the contents of the Hint menu can be changed during the course of the game. This helps prevent "spoilers." For instance, if the player consults the hints early in the game, it would be a terrible spoiler to find a hint listed as "How do I get the jewels out of the stomach of the shark?" before the player has encountered a shark or knows about the jewels. Using Adaptive Hints, you can cause that hint to appear only after the player has reached a point in the game where it makes sense for the hint to be available.

Special Features in Glulx

In **Chapter 21** of the Documentation, "Figures, Sounds and Files," you'll find a good discussion of how to add graphics files (digital photos or clip art in .jpg or .png format) to your game. Displaying graphics in a game can be a good way to give the game an extra dimension — but not all IF interpreters will display graphics. Some interpreters are *text-only.* In addition, some IF players are blind or visually impaired. One of the reasons these folks like IF is because they can play the games using screen-reader software. For these reasons, it's probably not a good idea to write a puzzle in such a way that an essential clue is found only in a graphic image. Think of graphics as enhancements.

The same chapter in the Documentation also discusses including sound files briefly. But again, you'll be at the mercy of the interpreter the player happens to be using. Support for sound playback in IF is generally not too reliable, so I'd suggest not worrying too much about this feature.

The glulx game format also supports opening several windows at once within the interpreter's main window. With this feature, you could display graphics in one sub-window and have the text game running in another sub-window. Instructions on how to use this feature are beyond the scope of this *Handbook.*

The current version of the Glulx specification is very limited when it comes to text styles, but this may change in the future.

What Does Inform 6 Have to Do with Inform 7?

In this *Handbook*, I've deliberately avoided getting too far into the deep end of Inform programming. If you work with Inform 7 for a while, though, you'll start to see occasional mentions of Inform 6.

In a technical sense, Inform 7 is built "on top of" Inform 6. Inform 6 is still being used behind the scenes when your game is compiled. But this fact is normally well hidden from the I7 programmer, and there's no reason why you need to concern yourself with it.

Once in a while, an expert programmer will be trying to achieve an effect that can only be achieved by including some I6 code in an I7 game. If you look at the code in a few Extensions written by experts, you'll probably see Inform 6 peeking through the curtain. Here's a line from a couple of Extensions by Emily Short — the first two I chose at random:

```
Use sequential action translates as (- Constant SEQUENTIAL_ACTION; -).
```

The parentheses and hyphens are what tells the compiler "Here's some Inform 6 code." What that particular line does … I don't know, and it doesn't matter. For more details on how Inform 6 code can be included in Inform 7 code, you can look at **pages 24.13** ("Using Inform 6 within Inform 7") through **24.25** ("Inform 6 adjectives") of the Documentation. But the things you can do with I6 code are far beyond the scope of this *Handbook*.

Think of it this way: Writing a game in Inform 7 is like driving a car down the road. Adding I6 code to an I7 game is like pulling over, popping the hood, and tinkering with the fuel pump. Once in a while, that may be the only way to get where you want to go, but you shouldn't need to do it too often, and nobody but an expert mechanic should try it at all.

So, Is Inform 7 the Ultimate IF Programming Language?

Given the popularity of Inform 7 and the ease with which newcomers can use it to start creating their own interactive fiction, it wouldn't be surprising if a lot of people get the impression it's the best IF development system that could ever be imagined. It would be wrong for this *Handbook* to draw to a close without commenting on that impression.

For all its strengths, Inform has some surprising weaknesses. A few of them have been touched on in the course of this book. Obviously, I feel that Inform is a very good language with which to approach writing interactive fiction, especially if you're new to computer programming. But my enthusiasm has more to do with Inform's approachability — its "natural language" syntax and its cross-platform IDE — than with the nuts and bolts of its design.

Some of its limitations seem to have arisen out of Graham Nelson's desire to preserve backward compatibility with older IF systems — not just Inform 6 but earlier systems dating back to the 1980s, when Infocom was still an active company releasing new text-based games. The fact that a game's release number (displayed in the banner at the beginning of the game) can't have decimal places, but can only be an integer, seems to have no rational basis other than the fact that that's how Infocom did it. Today, the standard in version numbering often includes not one but two decimal points — version 1.5.2, for instance. Inform just plain can't number your game's version that way.

To be sure, that's a trivial issue. A more serious example is the separation between .z5 and .z8 game files on the one hand and Glulx files on the other. The Z-machine format is an artifact of a bygone era. There's no real reason why it should still be supported today, except for the large installed base of Z-machine interpreters. On the other hand, the Z-machine is actually more powerful than a Glulx interpreter in a few restricted circumstances having to do with type styles; Glulx is, at this writing, still rather poor at allowing the author to customize type styles.

Other limitations or odd design choices seem to have arisen due to the piecemeal fashion in which Inform was developed. Because its syntax is very unlike the syntax of any other programming language, and because Nelson wanted the code to read as much like English as possible, he seems to have made some choices along the way that were less systematically consistent than *ad hoc*. If you've created a backdrop called the sky, for instance, you'll find that in some code statements you can refer simply to "the sky", while in others you have to refer to "the sky backdrop."

Another good example of this peculiarity — and I can think of half a dozen (let's not talk about the syntax for accessing the data in tables) — is found in the phrase "remove the banana from play." You might expect, based on how Inform generally works, to be able to say, "now the banana is nowhere." But that phrase won't compile. "play" is not an object in your game's code space, so being required to refer to "play" in order to banish an object is simply peculiar — especially when you consider that it's possible to *test* whether the banana is nowhere.

The fact that doors can't be moved probably simplifies the way Inform works internally, but in terms of the limitations it places on the author, it was a poor design choice. No more need be said about that.

Another limitation, and one that is of concern to a lot of authors, is that Inform insists that all of the code for your game be stored in a single file. No other modern programming language operates this way; the norm among programmers is to store the code for a single project in several files, which can be edited (and compiled) separately. There are several reasons why forcing all the code for a game to live in a single file was a poor design choice: It makes collaborating with other authors more difficult, it makes editing more difficult, and so on. The fact that the code file is always called story.ni is also a problem, because it makes the process of archiving successive versions of the project during development a bit more cumbersome and error-prone.

At the code design level, Inform doesn't allow multiple inheritance. For instance, a single object can't be both a device and a supporter. Lacking the ability to create such an object, the author has to choose a workaround in order to create as standard an in-game object as a stove. The workarounds are not, in most cases, burdensome. Usually, you can make one object a part of another object, and you'll be ready to move on. And Inform's device class is so bare-bones that there's almost no reason to use it at all. But the absence of multiple inheritance can occasionally force the author to perform more elaborate gyrations.

If you want to really master the deepest level of Inform 7 programming, you'll have no choice but to learn an entirely new and much more abstract set of code syntax: Inform 6. Learning two separate coding systems means extra work. To be sure, Inform is not the only computer language that works this way: Musicians who use Csound, for instance, can get at some of its deeper features only by learning to program in Python or C. But most interactive fiction authoring systems are not bifurcated in this way. TADS 3, for instance, can be entirely customized by writing new code that uses exactly the same syntax you would use to write your game (though the extensive use of macros and templates in game programming somewhat obscures this fact). TADS has a much more extensive built-in library than Inform, implements multiple inheritance, handles type styles, clickable hyperlinks, and multimedia — and as an added bonus, if you start learning programming with TADS, you'll find that the coding skills you develop will be much more readily transferrable to mainstream programming languages like Java and C++.

In sum — no, Inform 7 is not the be-all or end-all. It's flawed in some non-trivial ways. Yet at the same time, there's no denying that it's powerful, popular, well-supported, and quite easy to use, and has a number of terrific features. For many aspiring authors, it will be absolutely the right choice. Because it's still being developed, we can hope that over time, some of the issues mentioned above will be dealt with. And finally, the developers of Extensions for Inform have added, and continue to add, some powerful and unexpected capabilities. If you're new to writing IF, you need have no fear that in choosing Inform 7, you're painting yourself into a corner or headed down a dark alley. You can have confidence that you're making a wise choice, that you'll be able to produce games of amazing complexity and high quality.

All it takes is thick slabs of inspiration and months of hard work, coupled with generous amounts of head-scratching, hair-pulling, and teeth-grinding. Really, you hardly ever need to throw your computer across the room in sheer frustration. Once in a while, you may notice that you're actually having fun. And that's the point, isn't it?

Appendix A: Glossary

After: One of the rulebooks used in processing player commands. (See p. 130.)

backdrop: A special kind of object that can be in many rooms at once. (See p. 66.)

Before: One of the rulebooks used in processing player commands. (See p. 130.)

beta-testing: The second stage in testing a piece of software (such as a game) before it's released. (See p. 30.)

bug: An error in your code that causes it not to work the way you want it to. (See p. 28.)

Carry Out: A set of rulebooks used in processing player commands. Each action has its own Carry Out rulebook. (See p. 130.)

Check: A set of rulebooks used in processing player commands. Each action has its own Check rulebook. (See p. 130.)

command prompt: The symbol that appears in the game at the beginning of the line where the player is supposed to type commands. (See p. 19.)

comment: Material that's in your source code but that is ignored by the compiler. In Inform, comments are surrounded with square brackets [like this].

compiler: The software "machine" within Inform that turns your source code into a playable game. (See p. 16.)

container: A thing that the player can put other things into. Like a door, a container can be openable or lockable. (See p. 86.)

default: A default is what Inform does automatically, unless you tell it to do something different.

device: A thing that can be switched on or switched off. (See p. 116.)

door: A special kind of object that connects two rooms and that can be opened, closed, locked, and unlocked.

Extension: A file you can download that will add extra features to the Inform programming language. Extensions contain some Inform source code and also some documentation that tells you how to use the new features in the Extension. (See p. 35.)

Glulx: A game format suitable for large Inform games and those that include graphics and sound. Glulx games usually have filenames that end with .ulx or .blorb.

holdall: A special type of portable container that the player can carry. If the player has a limited carrying capacity, an Inform game can automatically deposit excess objects in the holdall. (See p. 101.)

IDE: Integrated Development Environment. The Inform 7 authoring system (which includes source code editing, built-in Documentation, a compiler, an interpreter, and so on) is an IDE.

Index: A set of pages in the Inform program that give you quick access to just about everything in your game.

indexed text: A special type of text data that can be easily searched and changed by the game's code while the game is running. (See p. 249.)

Instead: One of the rulebooks used in processing player commands. (See p. 130.)

interpreter: A piece of software that can run an interactive fiction game. (See p. 22.)

kind: An abstract term used to create a class of similar things. (See p. 41.)

noun: A special term that refers to whatever object the player mentioned in a command. (See p. 79.) For instance, if the player's most recent command was PICK UP THE BALL, the value "the noun" will be set to the ball object.

parser: The software mechanism that reads the player's input during a game and figures out what the input means. (See p. 19.).

printed name: The name Inform will use in the game when it needs to refer to the object. Usually, the printed name is whatever you called the object when creating it, but the printed name can be changed.

procedural rule: A special type of Inform rule that, while the game is being played, temporarily suspends or shuts off one of the rulebooks that are used to construct the model world and make decisions about what the player can or can't do.

Recipe Book: The second book of Documentation in the Inform program. To read the *Recipe Book*, scroll down to the bottom of the Table of Contents for *Writing with Inform* (in the Documentation panel) and click on the link.

region: A related group of rooms within your model world. (See p. 65.)

Report: A set of rulebooks used in processing player commands. Each action has its own Report rulebook. (See p. 130.)

room: A location (either indoors or outdoors) in which some part of the game takes place. (See p. 39.)

rule: An instruction that tells Inform what to do in a certain situation. Rules are gathered together by the compiler into rulebooks.

scene: A portion of your game that is structured so as to happen during a certain period of time. (See p. 202.)

scenery: A property that can be applied to things (including doors, supporters, and so on). When a thing is scenery, it will not be mentioned automatically in the game when the room description is printed; your room description will have to supply all mentions of scenery. Scenery can't be picked up and carried around by the player. (See p. 41.)

scope: The portion of the model world that is currently available to the player. Usually the scope is the same as the room the player is in, but in some special situations it may be different. (See p. 38.)

second noun: A special term that refers to the second object the player mentioned in a command. For instance, if the player's most recent command was PUT THE BALL IN THE BOX, the value "the second noun" would refer to the box object.

Skein: A panel in the Inform 7 IDE that keeps track of and can replay all of your testing sessions with your project (or at least a large number of recent sessions).

source code: Also called source text — what you write in the Source panel of Inform's IDE.

status line: The strip across the top of the window in a running game. Various types of information, such as the room name, score, and time of day within the game, can be displayed in the status line. (See p. 199.)

supporter: A kind of object in an Inform game. A supporter is an object like a table, that other objects in the game can be placed *on*. (See p. 86.)

switch: A group of lines (see p. 263). Inform will choose one line from the group depending on the current value of some variable.

table: A data structure in which Inform can store large amounts of related data in rows and columns. (See p. 241.)

thing: The basic kind of object in an Inform game. Doors, supporters, containers, and even people are all things, but the term "thing" is sometimes used to refer specifically to movable objects that aren't of any other kind.

truth state: A kind of variable that can have only one of two values — true or false. (See p. 225.)

variable: A value that changes while the game is running. A variable can be a number or an object, for example. (See p. 224.)

virtual machine: A piece of software that creates a sort of "box" inside which other software (such as an Inform game) can run. It's called a "virtual" machine because it isn't an actual hardware machine.

visible thing: An object that is somewhere in the model world of the game, but not necessarily visible to the player at this moment. (See p. 141.)

wearable: A property of things. Wearable things can be worn by the player, or by other characters.

Z-code: A type of compiled game that is compatible with the Z-machine. Inform can produce Z-code games in the .z5 and .z8 formats.

Z-machine: An interpreter that can run games written in Inform. (Large games and games with special features such as graphics are written using Inform's Glulx format, which is not compatible with the Z-machine.) Z-machine interpreters are available for most computer platforms, including many obsolete operating systems and some portable devices.

Appendix B: Short Sample Games

In this appendix are included a few miniature games that illustrate various techniques. These games are a bit too complex to fit well within the main flow of the *Handbook,* so they're gathered here instead, in no particular order.

Flea Market

We'll start with a simple scenario that was suggested by Jay in a post on the newsgroup. Jay's goal was to make NPCs respond at random to items carried by the player. Each time the player takes inventory, one of the other shoppers in the flea market will notice some random thing the player carries, and comment on it. Jay was trying to do this using tables, but I couldn't figure out how to make that work, so I tried a different approach.

This example illustrates four techniques: calling a function, creating a definition, using what computer programmers call a *switch* statement, and using a while loop.

Inform's switch statement (see **p. 11.8** of the Documentation, "Otherwise") uses a double hyphen to indicate the various entries in the switch block. Only one of the lines in a switch block will be run at any given time. In this case, we enter the switch block by selecting a random number between 1 and 5 and then choosing a say statement based on the value of the random number.

We're creating a function ("To say the desire") and passing it two arguments — a person and a thing. Within the function, we refer to the person as P and the thing as J. There's nothing special about these symbols; we could just as easily have written "To say the desire of (dude - a person) for (stuff - a thing)" and then referred to "[dude]" and "[stuff]" in the say statements.

```
To say the desire of (P - a person) for (J - a thing):
    if a random number from 1 to 5 is:
    -- 1: say "'Gee, I'd sure like to have a [J] like that,' [P] remarks.";
    -- 2: say "'That's sure a swell-looking [J],' says [P].";
    -- 3: say "[P] edges closer to you and says wistfully, 'I've been looking for years for a [J].'";
    -- 4: say "[P] hovers over you covetously. 'Say, were there more [J]s where you found that?'";
    -- 5: say "'Wow, that's a really nice [J],' [P] says enviously."

After taking inventory:
    if nothing is held by the player or the player is alone:
        rule succeeds;
    otherwise:
        let shopper be the player;
        while shopper is the player:
            let shopper be a random person in the location;
        let junk be a random thing held by the player;
        say "[the desire of shopper for junk]".
```

```
Definition: a person (called P) is alone if the number of persons in the location of P is 1.
```

Before entering the while statement, we define a variable called shopper, and give it the value of the player. When the while statement starts, shopper is the player. The while statement is a loop. It will execute over and over, randomly picking one person after another in the location, until the person it picks is *not* the player. At that point, the condition (while shopper is the player) becomes false, the loop ends, and the rest of the After rule runs. The rule chooses a random thing held by the player, combines it with the randomly chosen NPC, and says the desire of the random person for the random thing.

The definition is simple: We need to tell Inform what we mean by “alone,” because the “After taking inventory” rule needs to check this. Can you see what will happen if the “After taking inventory” rule doesn’t include the condition “if … the player is alone”? If the player ever takes inventory when no one else is in the location, the rule will go into an endless loop. The while statement will keep trying forever to find an NPC in the location.

Here’s the rest of the game:

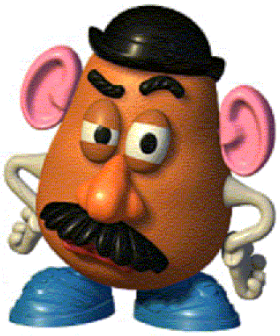
```
The Flea Market is a room. "All manner of things are for sale here, lined up in rows on folding tables."

A folding table is a supporter in the Flea Market. On the table are a replica Maltese Falcon, a phallic fertility statue of Aziz, a 1956 Ford carburetor, a headless Barbie doll, and a broken Pez dispenser.

Wanda is a woman in the Flea Market.
Julianne is a woman in the Flea Market.
Chrissie is a woman in the Flea Market.

The Parking Lot is north of the Flea Market. "Your car is parked here."

Test me with "i / take pez and ford and falcon and doll / i / i / n / i".
```

Since we don’t know what the player may be carrying or who may be in the location, we pretty much have to choose NPCs and held items at random. However, when items are selected at random, Inform will sometimes select the same item several times in a row. This can result in a repeating output, which will make the game seem wooden. If you’d prefer to avoid this effect, you can step through the list of entries in the “To say the desire” rule in a repeating order, like this:

```
The desire-index is a number that varies. The desire-index is 0.

To say the desire of (P - a person) for (J - a thing):
	increase the desire-index by 1;
	if the desire-index is greater than 5:
		now the desire-index is 1;
	if the desire-index is:
	-- 1: say "'Gee, I'd sure like to have a [J] like that,' [P] remarks.";
	-- 2: say "'That's sure a swell-looking [J],' says [P].";
	-- 3: say "[P] edges closer to you and says wistfully, 'I've been
looking for years for a [J].'";
	-- 4: say "[P] hovers over you covetously. 'Say, were there more [J]s where you found that?'";
	-- 5: say "'Wow, that's a really nice [J],' [P] says enviously."
```

Here, we've created a global variable (desire-index). It's global so that it will persist throughout the game; a local variable within a rule will be destroyed when the rule finishes. We then increase the desire-index by 1 each time the rule runs and check to see if it has gone past 5. If so, we reset it to 1. Now the five statements will be used in the order shown, over and over (but with random people and random things).

Mr Potato Head



The next example game illustrates several useful points. First, it shows how to make parts that can be attached to or detached from an object. (In this case, the object is the ever-obliging Mr Potato Head.)

```
Include Plurality by Emily Short.

The Potato Factory is a room.

Mr Potato Head is a man in the Factory. The description is "Mr Potato head is big and round and brown[attachment-list]."

To say attachment-list:
    let count be 0;
    repeat with item running through things that are a part of Mr Potato Head:
        increase count by 1;
    if count is 0:
        do nothing;
    otherwise:
        if the lips are part of Mr Potato Head:
            increase count by 1;
        say ". Adding an attractive facial character to Mr Potato Head [if count is greater than 1]are[otherwise]is[end if] [a list of things that are part of Mr Potato Head]".
```

The point of the "say attachment-list" code is that we want to append a list of Mr Potato Head's features to his description. Just to make things slightly more interesting, the lips are plural-named, so if the lips are part of Mr Potato Head we need to increase the count by 1 in order to print out "are" rather than "is" if the lips are the only feature that has been attached.

Next, we'll create some facial attachments, and also an irrelevant item (the banana) for testing purposes.

```
A facial attachment is a kind of thing. The nose is a facial attachment. The left eye is a facial attachment. The right eye is a facial attachment. Some lips are a facial attachment.

The player carries the nose. The player carries the left eye. The player carries the right eye. The player carries the lips.

The player carries a banana.
```

The command the player is most likely to try is of the form PUT NOSE ON MR POTATO HEAD. But as far as Inform is concerned, putting it on is a different action from tying it to, so we need to redirect the putting it on action using an Instead rule. The next Instead rule handles both the error-checking for the adding of facial attachments, and the action.

```
Instead of putting something on Mr Potato Head:
	try tying the noun to Mr Potato Head.

Instead of tying something to Mr Potato Head:
	if the noun is not a facial attachment:
		say "That's not something you can attach to Mr Potato Head.";
	otherwise if the noun is part of Mr Potato Head:
		say "[The noun] [is-are] already attached to Mr Potato Head.";
	otherwise:
		now the noun is part of Mr Potato Head;
		say "You attach [the noun] to Mr Potato Head."
```

In creating our new detaching it from action, we also need to consider that the player's most likely command is simply TAKE NOSE. The parser's default response would be, "That seems to be a part of Mr Potato Head." Inform very sensibly doesn't let the player go around removing the parts of things. So we need another Instead rule to map TAKE NOSE onto the detaching it from action.

```
Detaching it from is an action applying to two things and requiring light. Understand "detach [something] from [something]" and "remove [something] from [something]" as detaching it from.

Check detaching it from:
	if the noun is not part of the second noun:
		say "But [the noun] [is-are] not attached to [the second noun]." instead;
	otherwise if the noun is not a facial attachment:
		say "[The noun] do[es] not appear to be detachable." instead.

Carry out detaching it from:
	now the player carries the noun.

Report detaching it from:
	say "You detach [the noun] from [the second noun]."

Instead of taking something:
	if the noun is part of a thing (called the owner):
		try detaching the noun from the owner instead;
	otherwise:
		continue the action.
```

And finally, just for fun, we'll add a victorious outcome to the game:

```
Every turn:
	if the left eye is part of Mr Potato Head and the right eye is part of Mr Potato Head and the lips is part of Mr Potato Head and the nose is part of Mr Potato Head:
		say "'Oh, thank you!' cries Mr Potato Head. 'You've restored my faith in human nature!'";
		end the game in victory.
```

A Dangerous Jewel Box

The next example shows a way of printing out a single message when the player tries to take several items and is prevented from doing so. There are three jewels in the jewel box, and if the box is in its dangerous state, we want the player character’s fingers to be zapped by an electric shock if he tries to get the jewels from the box. But what we don’t want is an output that looks like this:

```
>take jewels
ruby: Zap! As your fingers near the jewel box, you recoil from a powerful electric shock.
diamond: Zap! As your fingers near the jewel box, you recoil from a powerful electric shock.
sapphire: Zap! As your fingers near the jewel box, you recoil from a powerful electric shock.
```

This would be silly, because the player character would naturally stop after being shocked the first time. Getting the output to read nicely turns out to be complicated, because there are many possible conditions that might occur in the game. The box might be safe rather than dangerous (this is handled in the example with the command TURN CARVING). The player might make the box safe, take two jewels, drop them on the floor, make the box dangerous again, and then try TAKE JEWELS. What should happen then?

The extension Consolidated Multiple Actions by John Clemens can also handle this type of scenario — but it only works in Glulx games. If you don’t need or want all of its functionality, studying the code below may show you how to handle this type of situation.

The code below contains comments that explain its logic.

```
The Sultan's Treasure Room is a room. "Awesome treasures surround you! To the north, an arch opens on a balcony."

[To make sure the scoping works properly, we'll add a second room so the tester can carry a jewel or two elsewhere, drop them, and come back:]
The Balcony is north of the Treasure Room. "From here you can see the entire city. The treasure room is back to the south."

The jewel box is an open container in the Treasure Room. "A jewel box sits here, invitingly open. On the front of the box is an intricate ivory carving. In the box you can see [a list of things in the jewel box]." The jewel box can be safe or dangerous. The jewel box has some text called the zap-message. The zap-message of the jewel box is "Zap! As your fingers near the jewel box, you recoil from a powerful electric shock". The intricate ivory carving is part of the jewel box.

[We can make the jewel box safe or dangerous with 'turn carving':]
Instead of turning the carving:
    say "Click -- the carving rotates a quarter-turn to the [run paragraph on]";
    if the jewel box is dangerous:
        now the jewel box is safe;
        say "left.";
    otherwise:
        now the jewel box is dangerous;
        say "right."

A gem is a kind of thing. Understand "jewel" as a gem. Understand "jewels" and "gems" as the plural
of gem.

The diamond is a gem in the jewel box. The ruby is a gem in the jewel box. The sapphire is a gem in
the jewel box.

[The box only protects jewels. The token can be taken from it at any time:]
The player carries a subway token.

A procedural rule: if taking gems when the jewel box is dangerous then ignore the announce items
from multiple object lists rule.

[We need to cover the action of removing it from as a separate procedural rule:]
A procedural rule: if removing gems from the jewel box when the jewel box is dangerous then ignore
the announce items from multiple object lists rule.

Before taking gems:
    let L be the multiple object list;
    let N be the number of entries in L;
    [If the player is only trying to take one jewel, the length of the multiple object list will be 0:]
    if N is 0:
        if the noun is in the jewel box and the jewel box is dangerous:
            say "[zap-message of the jewel box]." instead;
        otherwise:
            continue the action;
    [Still here? Then either the player is trying to take several jewels or the jewel box is safe:]
    if the jewel box is safe:
        continue the action;
    [Okay, now we know the jewel box is dangerous, but we don't yet know whether any of the
jewels the player aims to take are actually IN the jewel box ... so we'll set up a flag and a list:]
    let danger-present be a truth state;
    let danger-present be false;
    let safe-list be a list of things;
    let safe-list be {};
    [Now let's find out what the player is trying to take:]
    repeat with G running through gems:
        if G is listed in L:
            if G is in the jewel box:
                now danger-present is true;
            otherwise:
                add G to the safe-list;
    [If none of the jewels the player is trying to take is in the dangerous box:]
    if danger-present is false:
        continue the action;
    [Now we know that at least one of the jewels in the multiple object list is in the dangerous box
-- but maybe some of them aren't in the box. In that case, we're going to assume there is no other
obstacle, such as inventory management, a greedy hyena, or the player wearing mittens, that would
prevent their being picked up. We'll only check to make sure they're in the location:]
```

```
    if the number of entries in safe-list is not 0:
        repeat with item running through safe-list:
            if item is enclosed by the location and item is not carried by the player:
                now the player carries item;
                say "[item]: Taken.";
    truncate L to 1 entries;
    alter the multiple object list to L;
    say "[zap-message of the jewel box]." instead;

Test me with "put token in box / take token and ruby / take jewels / take diamond / turn carving / take diamond and ruby / turn carving / drop all / take jewels".
```

The heavy lifting is done by the "Before taking gems" rule. Note that if one or more gems are available for taking, this rule has to handle the process manually. It can't include, near the end, the line "try taking item", because that will cause Inform to try running the entire rule again. It will produce a loop, which will result in a run-time error.

This example uses two **procedural rules.** Procedural rules are an advanced concept in Inform programming. You can learn a bit about them by reading **p. 18.13** of the Documentation, called (logically enough) "Procedural rules."

The Omega Machine

This sophisticated example showed up recently on the intfiction.org forum. The author, who goes by the handle SJ_43, was responding to a question from someone named Eudoxia about how to make a device that the player can give commands to — for instance, a command like COMPUTER, CHECK THE STABILIZERS. The game below doesn't implement anything quite that complicated, but it provides a framework with which you could easily do it. The sneaky Inform 6 trick in this code is the line, "Include (- has talkable, -) when defining a computer." The parentheses and dashes are used (as described on p. 257) to drop the code out of I7 and down into I6. "has talkable" is I6 code that means, more or less, "This is an object the player may want to be able to talk to."

Note that the Default Messages Extension may change or not be compatible with a future version of Inform 7. It's used here to create a special error message for devices of the computer kind.

I've customized SJ_43's code by adding some calendar messages for Omega. In a real game, you'd probably want to store such messages in a table, so as to be able to refer to different messages at different points in the game.

```
Include Default Messages by David Fisher.

A computer is a kind of device.
Include (- has talkable, -) when defining a computer.

Checking for mail is an action applying to nothing.
Understand "mail" as checking for mail.
```

```
Check an actor checking for mail:
	unless the actor is a computer, stop the action.

Carry out a computer checking for mail:
	say "'You have no messages.'"

Displaying the calendar is an action applying to nothing.
Understand "calendar" as displaying the calendar.

Check an actor displaying the calendar:
	unless the actor is a computer, stop the action.

Carry out a computer displaying the calendar:
	say "'No appointments today.'"

Checking for mail is doing computer stuff. Displaying the calendar is doing computer stuff.
Persuasion rule for asking a computer to try doing computer stuff: persuasion succeeds.

Instead of doing computer stuff, say "That command is for computers only."

Instead of a switched off computer (called comp) doing something:
	say error message for the comp;
	rule succeeds.

Asking a computer about something is computer conversation.
Telling a computer about something is computer conversation.
Asking a computer for something is computer conversation.
Instead of computer conversation, say "You can only do that to something animate."
Instead of answering a computer that something, say error message for the noun.

To say error message for (comp - a computer):
	if comp is switched on, say "'Does not compute!'";
	otherwise say "[The comp] is currently switched off."

Table of custom library messages (continued)
Message Id	Message Text
LibMsg <person ignores command>	"[if the main object is a computer][error message for the main object][otherwise][The main object] has better things to do.[/p][end if]"

The Lab is a room. Omega is a computer in the Lab. Understand "pda" as Omega. The description of omega is "Your trusty handeheld pda.  The chrome-plated device is voice-activated, and no batteries are necessary since it has a built-in microfusion reactor."

Carry out Omega displaying the calendar:
	say "[one of]'You're late for your appointment with the white rabbit!' says Omega[or]'Time for a flu shot!' says Omega[or]'I'm afraid I can't do that, Dave,' says Omega[stopping].";
	rule succeeds.

Test me with "take omega / omega, mail / turn on omega / omega, mail / omega, calendar / g / g / omega, sing / mail".
```

The Lurking Critter

This example, which was inspired by an answer that Mike Gentry gave on the newsgroup to a question from S. John Ross, is included principally to show a way of changing the state of an object (the sword) and of reporting that change in certain conditions — namely, if the player is carrying the sword. The idea that the sword glows if the lurking critter is nearby is borrowed from Zork. Incidentally, one of the brightnesses is called “glowless” because Inform had trouble with a brightness called “not glowing.” The “not” is too confusing to the compiler.

We’ll start by creating a checkerboard-like matrix of 16 rooms. This matrix is at least mildly interesting in that the room connections cross one another without intersecting. That is, you can go southeast from Room1 to Room6, or southwest from Room2 to Room5, but the two paths are independent. Both the player and the lurking critter can move diagonally from room to room as well as orthogonally. The critter will move at random, but only 50% of the time.

```
Room1 is a room. Room2 is east of Room1. Room3 is east of Room2. Room4 is east of Room3.

Room5 is south of Room1. Room6 is east of Room5 and south of Room2. Room7 is east of Room6 and south of Room3. Room8 is east of Room7 and south of Room4.

Room9 is south of Room5. Room10 is east of Room9 and south of Room6. Room11 is east of Room10 and south of Room7. Room12 is east of Room11 and south of Room8.

Room13 is south of Room9. Room14 is east of Room13 and south of Room10. Room15 is east of Room14 and south of Room11. Room16 is east of Room15 and south of Room12.

Room5 is southwest of Room2. Room6 is southwest of Room3 and southeast of Room1. Room7 is southwest of Room4 and southeast of Room2. Room8 is southeast of Room3.

Room9 is southwest of Room6. Room10 is southwest of Room7 and southeast of Room5. Room11 is southwest of Room8 and southeast of Room6. Room12 is southeast of Room7.

Room13 is southwest of Room10. Room14 is southwest of Room11 and southeast of Room9. Room15 is southwest of Room12 and southeast of Room10. Room16 is southeast of Room11.

The lurking critter is a man in Room4.

Brightness is a kind of value. The brightnesses are glowing brightly, glowing faintly, and glowless.

The player carries a sword. The sword has a brightness. The brightness of the sword is glowless. The description is "A trusty critter-sensing weapon[if the brightness of the sword is glowing faintly]. It's glowing faintly[otherwise if the brightness of the sword is glowing brightly]. It's glowing brightly[end if]."
```

This next routine does two things. It always adjusts the sword-glow to reflect the sword’s current proximity to the lurking critter. It then reports a change in the sword-glow — but only if the player carries the sword. I chose to do it this way because in the real world, you’d be less likely to notice a change if the sword were simply lying nearby than if it’s in your hand.

```
To adjust sword-glow to (b - a brightness):
	let current glow be the brightness of the sword;
	now the brightness of the sword is b;
	if b is the current glow:
		do nothing;
	otherwise if b is glowless:
		if the player carries the sword:
			say "Your sword is no longer glowing.";
	otherwise if b is glowing faintly:
		if the player carries the sword:
			say "Your sword glows with a faint blue glow.";
	otherwise:
		if the player carries the sword:
			say "Your sword is glowing brightly."
```

This next definition is used, rather than the library's default terminology, "adjacent room," because room adjacency isn't considered to exist if two rooms are separated by a door.

```
Definition: a room is neighboring if the number of moves from it to the location is 1.
```

And finally, we'll move the critter (maybe) and adjust the sword-glow:

```
Every turn:
	if a random chance of 1 in 2 succeeds:
		let current location be the location of the lurking critter;
		let next location be a random room which is adjacent to the current location;
		if the lurking critter is visible:
			say "The critter [one of]slouches[or]slithers[or]shambles[or]lurches[at random] away.";
		move the lurking critter to next location;
		if the lurking critter is visible:
			say "A critter [one of]oozes[or]staggers[or]ambles[or]creeps[at random] into the room.";
	[Whether or not the critter has moved, we need to adjust the sword-glow, because the player may have moved.]
	if the lurking critter is in the location:
		adjust sword-glow to glowing brightly;
	otherwise if the lurking critter is in a neighboring room:
		adjust sword-glow to glowing faintly;
	otherwise:
		adjust sword-glow to glowless.
```

Restraints

Preventing the player from performing certain actions in certain situations is a standard type of puzzle. For instance, if the player is wearing handcuffs, picking things up (that is, using the TAKE or GET command) probably shouldn't work. I'll leave it up to your imagination to figure out how the player character might be able to remove a pair of handcuffs while unable to pick up the key. In the example below, a simple workaround is used — the player is allowed to pick up the key, but nothing else.

This example, which was posted by Zeborah on the forum at intfiction.org and then revised by me with a little help from Mike Tarbert, provides a general-purpose way to intercept actions depending on what sort of restraint the PC is wearing. It uses a table to correlate actions with restraints.

```
A restraint is a kind of thing. A restraint is usually wearable. The blindfold, the gag, the ball-and-chain,
and some handcuffs are restraints.

Instead of doing something when the player wears a restraint:
    repeat through the Table of Restricted Movements:
        if the player wears the restraint entry:
            let impulses be a list of action-names;
            let impulses be the movement entry;
            if the action-name part of the current action is listed in impulses:
                say "You have no hope of [the current action] while wearing [the restraint
entry]." instead;
    continue the action.

The handcuffs can be locked or unlocked. The handcuffs are locked.

Instead of locking the handcuffs with the small iron key:
    if the handcuffs are locked:
        say "They're already locked.";
    otherwise if the player does not carry the small iron key:
        say "You don't seem to have the key.";
    otherwise:
        now the handcuffs are locked;
        say "You lock the handcuffs with the small iron key."

Instead of taking off the handcuffs:
    if the handcuffs are locked:
        say "The handcuffs seem to be locked.";
    otherwise:
        now the player carries the handcuffs;
        say "You remove the handcuffs."

Instead of wearing the handcuffs:
    if the handcuffs are worn:
        say "You're already wearing the handcuffs.";
    otherwise if the handcuffs are locked:
        say "You'll need to unlock the handcuffs before you can put them on.";
    otherwise:
        now the player wears the handcuffs;
        say "You put on the handcuffs."

Instead of unlocking the handcuffs with the small iron key:
    if the handcuffs are unlocked:
        say "But they're not locked!";
    otherwise if the player does not carry the small iron key:
        say "You don't seem to have the key.";
```

```
	otherwise:
		now the handcuffs are unlocked;
		say "It's awkward, but you manage to insert the key in the keyhole and turn it. Now the handcuffs are unlocked."

Instead of taking the small iron key when the player wears the handcuffs:
	now the player carries the small iron key;
	say "You manage to pick up the key, even though you're wearing handcuffs."

Instead of opening the tool chest when the player wears the handcuffs:
	if the tool chest is open:
		say "It's already open.";
	otherwise:
		now the tool chest is open;
		say "It's awkward, but you manage to open the tool chest while wearing the handcuffs."

The Dungeon is a room. "A dank and dismal dungeon. You can go north."

The Dismal Crypt is north of the Dungeon. "The walls of the crypt are oozing...."

The player wears the ball-and-chain and the handcuffs.

The tool chest is an openable container in the Dungeon. The tool chest is closed. The blindfold and gag are in the tool chest. The small iron key is in the tool chest.

Table of Restricted Movements
restraint	movement
ball-and-chain {going action, jumping action, entering action, exiting action, getting off action, climbing action}
gag	{eating action, kissing action, answering it that action, telling it about action, asking it about action, asking it for action, saying yes action, saying no action, tasting action, saying sorry action, singing action}
handcuffs	{taking action, removing it from action, taking off action, dropping action, putting it on action, inserting it into action, searching action, touching action, waving action, pulling action, pushing action, turning action, squeezing action, eating action, consulting it about action, locking it with action, unlocking it with action, switching on action, switching off action, opening action, closing action, wearing action, attacking action, showing it to action, throwing it at action, cutting action, tying it to action, drinking action, swinging action, rubbing action, setting it to action}
blindfold	{looking action, searching action, examining action}

Test me with "n / remove ball-and-chain / remove handcuffs / open chest / unlock handcuffs with key / remove handcuffs / remove ball-and-chain / n".
```

This example takes advantage of the fact that when the Inform compiler builds rulebooks, it lists more specific rules ahead of more general rules. The Instead rules for the tool chest, key, and handcuffs will be listed before the general-purpose Instead rule, so they'll allow the player to solve the puzzle by opening the chest, picking up the key, and then unlocking the handcuffs.

The example also shows how to use a table containing list entries, and how to use the phrase "action-name part of the current action," which is not a syntax mentioned elsewhere in this *Handbook* (nor, for that matter, in the Documentation). It’s a useful syntax to know. For instance, if the current action is “going north”, the action-name part of the action is “going”.

After removing the handcuffs, you can experiment further with this example by wearing the blindfold and trying to examine things, or by adding an NPC, wearing the gag, and then trying to talk to the NPC.

Broken Eggs

This next example was originally written by Jason Travis and posted in the newsgroup, in response to a question from Rob Cowell about how to break a box full of eggs when the box was dropped. I’ve expanded the example quite a bit. It shows how to handle a collection of indistinguishable objects (the eggs) when any individual egg can be either broken or unbroken.

First, we’ll set up the game and let Inform know that an egg is a kind of thing. We’ll add some code that will allow eggs to be broken or unbroken. Note the use of a couple of “before printing” rules to let the player know what sort of egg is being mentioned. This is a handy alternative to changing the printed name of an egg object when the egg itself is in the process of getting broken.

```
"Humpty Dumpty Doesn't Live Here" by Jason Travis & Jim Aikin

The story headline is "A heart-rending tale of clumsiness and its irreversble consequences,"

The Kitchen is a room. "A well-appointed kitchen with all sorts of things you really don't need to
interact with."

An egg is a kind of thing. An egg can be unbroken or broken. An egg is usually unbroken. The
description is "[if unbroken]Ovoid and whitish.[otherwise]Yolk and other stuff grotesquely splashed in
an interesting pattern among various sharp bits of shell."

Understand the unbroken property as describing an egg.

Understand "sharp", "bits", "shell", "yellow", and "yolk" as broken. Understand "whole", "white",
"whitish", "ovoid", and "fresh" as unbroken.

Before printing the name of a broken egg, say "broken ".

Before printing the plural name of a broken egg, say "broken ".
```

We don’t want the player carrying around broken eggs, so we’ll get rid of them if the player tries to take them. This next bit of code illustrates the use of the word “called” to create a reference (BE) that we can use while writing the Instead rule. In this case, we could just as easily skip the “called” reference and say “remove the noun from play”, but “called” is a handy bit of syntax to know, so we’ll use it here.

```
Instead of taking a broken egg (called BE):
	say "You spend a good deal of time cleaning it up, throwing the gooey
debris into the trash.";
```

```
    remove BE from play.
```

Next, we need a box with some eggs in it. Note the use of the text substitution “[box-contents]”. This lets us write a whole To say rule that will mention the eggs in the box if there are any. By default, Inform doesn’t list the contents of an open container when the container is examined. But since broken eggs are highly noticeable, we’ll tack a list of things in the eggbox onto the description:

```
The eggbox is a closed openable container in Kitchen. The description is "It is made of recycled paper, and is decorated with a garish cartoon showing a line of happy chickens with their wings wrapped across one another's shoulders as they dance the can-can[box-contents]." The eggbox contains six eggs. Understand "box", "recycled", "paper", "happy", "chickens", and "carton" as the eggbox.

To say box-contents:
    if the eggbox is closed:
        do nothing;
    otherwise if the number of things in the eggbox is 0:
        do nothing;
    otherwise:
        say ". In the box you can see [a list of things in the eggbox]".
```

Now it’s time to break some eggs. The first part is easy. If the player uses the command DROP EGG, something bad will happen. We’ll break the egg in an After rule because we want to make sure the dropping action has been successfully carried out. If we used an Instead rule, we’d have to move the broken egg to the floor of the room manually, because the Instead rule would bypass Inform’s normal handling of the DROP action. We’d also have to make sure the player was actually carrying the egg, for the same reason.

The consequences of dropping the eggbox are more complicated. We’re going to assume that if the eggbox is closed, the eggs in it won’t break when the box is dropped. (Kids — don’t try this at home!) If the box is open, the eggs in it will break. But what if the box contains both some broken eggs and some unbroken ones? That could happen, for instance, if the player uses the commands OPEN BOX, TAKE EGG, DROP BOX, PUT EGG IN BOX, TAKE BOX. At this point the box would contain (probably) some broken eggs and an unbroken one.

```
After dropping an egg (called E):
    now E is broken;
    say "The egg tumbles to the floor in slow motion, breaking apart spectacularly. Somewhere in the distance you hear the forlorn cluck-cluck-cluck of a mother hen."

After dropping the eggbox:
    if the eggbox is closed:
        if the eggbox does not contain an egg:
            say "The empty box skitters across the floor.";
        otherwise if the eggbox contains a broken egg:
            say "The eggbox makes a sort of squishy, sloshing sound as it hits the floor.";
        otherwise:
            say "You hear [the number of eggs in the eggbox in words] egg[s] jouncing around safely in the carton.";
    otherwise:
        if the eggbox does not contain an egg:
            say "The empty box thunks hollowly on the floor.";
        otherwise if the eggbox contains an unbroken egg:
            say "Whoops! Broken [run paragraph on]";
            if the number of unbroken eggs in the eggbox is greater than 1:
                    say "eggs.";
            otherwise:
                    say "egg.";
            now every egg in the eggbox is broken;
        otherwise:
            let N be the number of broken eggs in the eggbox;
            say "The broken egg[if N is greater than 1]s slosh[otherwise] sloshes[end if]
around viscously in the eggbox as it hits the floor."
```

Studying this After rule may help you see how to structure blocks of code. The outermost if-test is “if the eggbox is closed … otherwise”. Within each of these blocks, we test whether the eggbox does not contain an egg, and deal with that possibility. And so on.

The use of “[run paragraph on]” and “let N be the number...” is also worth a quick look, if you’re not familiar with these tools.

Indoors & Outdoors

Inform’s built-in library creates rooms in a basic way. They have no walls, floor, or ceiling. If outdoors, they have no ground or sky. Such objects can easily be created as backdrops. A more serious limitation is that even outdoor rooms are, by default, “sealed containers.” You can’t look from one outdoor room into another and see anything that’s there.

My first thought was to bundle the example game below as an Extension, in order to provide these features and a few others for anyone who might find them useful. But I suspect authors will almost always want to customize the implementation in various ways. Since none of the code in the example game is tucked away in an Extension, you can easily copy it into your own game and modify it as needed.

If you want to write a realistic outdoor setting, spend some time studying page 3.4, “Continuous Spaces and the Outdoors,” in the Recipe Book. The examples on that page illustrate some useful techniques. For the present example, we’re going to implement both outdoor and indoor rooms. We’ll start by creating some backdrops and a couple of kinds of room. (Note that to use this code, you will usually need to add a few extra words to each room definition, to tell Inform whether the new room is an indoor room or an outdoor room.)

We need a special “check taking” rule for the sky, because by default Inform will respond to the player’s attempt to take a backdrop by saying “That’s hardly portable.” This response works fine for walls, floor, ceiling, and ground, but not for the sky.

```
Include Plurality by Emily Short.

An outdoor room is a kind of room. An indoor room is a kind of room.
```

```
The sky is a backdrop. The description is "A bright and cloudless blue."

Check taking the sky:
	say "You can't touch the sky.";
	rule fails.

The ground is a backdrop. The description is "The ground is a bit uneven." Understand "uneven" as the ground.
The ceiling is a backdrop. The description is "A few cobwebs up there." Understand "cobwebs" as the ceiling.
The floor is a backdrop. The description is "The floor is flat and level."

A backdrop has a direction called wall-direction. The wall-direction of a backdrop is usually nowhere.

A wall is a kind of backdrop.
The east wall is a wall. The wall-direction of the east wall is east.
The north wall is a wall. The wall-direction of the north wall is north.
The south wall is a wall. The wall-direction of the south wall is south.
The west wall is a wall. The wall-direction of the west wall is west.
The description of a wall is "It's vertical." Understand "walls" as the plural of a wall.

A corner is a kind of backdrop.
The northeast corner is a corner. The wall-direction of the northeast corner is northeast.
The northwest corner is a corner. The wall-direction of the northwest corner is northwest.
The southeast corner is a corner. The wall-direction of the southeast corner is southeast.
The southwest corner is a corner. The wall-direction of the southwest corner is southwest.
The description of a corner is "Two walls join here at right angles." Understand "corners" as the plural of a corner.
```

Defining some of the directions as diagonal will become important if the player tries LOOK NORTHEAST, for instance, while indoors. We want to refer to the corner of the room, not to a wall (since most likely there will be two walls meeting in the northeast). Having taken care of that detail, we create a pair of regions, in order to put the backdrops in them.

A significant limitation of Inform, which may come into play if you try adapting the code in this example for use in your game, is that regions can't overlap. (The basic way of handling region definitions is explained on p. 65.) Because of this limitation, if you use the code below you won't be able to include both an indoor room and an outdoor room in any of the regions you might want to define in your game. Indoor rooms are in the Great Indoors region and outdoor rooms in the Great Outdoors region.

```
A direction can be orthogonal or diagonal. A direction is usually orthogonal. Northwest is diagonal. Southwest is diagonal. Northeast is diagonal. Southeast is diagonal.

The Great Outdoors is a region. All outdoor rooms are in the Great Outdoors.
The Great Indoors is a region. All indoor rooms are in the Great Indoors.

The sky is in the Great Outdoors. The ground is in the Great Outdoors.
The floor is in the Great Indoors. The ceiling is in the Great Indoors. The east wall is in the Great Indoors. The north wall is in the Great Indoors. The west wall is in the Great Indoors. The south wall is in the Great Indoors.
The northeast corner is in the Great Indoors. The northwest corner is in the Great Indoors. The southeast corner is in the Great Indoors. The southwest corner is in the Great Indoors.
```

Next, we'll add some code to let the player use commands of the form LOOK EAST. This type of command will do one thing if the player is in an indoor room (it will examine a wall) and another if the player is in an outdoor room (it will use the new direction-looking action that we're about to define).

Note the use of the "listed instead of" line below to replace a default library rule with our own version. The library's default just prints out a message, but we need more nuance.

```
Carry out examining (this is the new examine directions rule):
    if the noun is a direction:
        if the location is an indoor room:
            repeat with item running through backdrops in the location:
                if the wall-direction of item is the noun:
                    try examining item instead;
        otherwise:
            try direction-looking the noun instead;
    otherwise:
        continue the action.

The new examine directions rule is listed instead of the examine directions rule in the carry out examining rulebook.

Looking toward is an action applying to one visible thing. Understand "look toward [any room]" as looking toward.

Chosen direction is a direction that varies.

Check looking toward:
    if the noun is the location:
        try looking instead;
    otherwise if the noun is not neighboring:
        say "You can't see that far." instead.

Carry out looking toward:
    change the chosen direction to the best route from the location to the noun;
    try direction-looking the chosen direction instead.

Direction-looking is an action applying to one visible thing and requiring light. Understand "look [direction]" as direction-looking.

Carry out direction-looking:
    let D be the noun;
    if the location is an indoor room:
        if D is inside or D is outside:
            say "You see nothing unusual.";
```

```
		otherwise if D is up:
			try examining the ceiling instead;
		otherwise if D is down:
			try examining the floor instead;
		otherwise if D is diagonal:
			say "The [D] corner of the room is an ordinary corner.";
		otherwise:
			say "The [D] wall looks like a wall.";
	otherwise:
		let R be the room D from the location;
		if R is a room:
			assemble the huge-list for R;
		otherwise:
			now the huge-list is {};
		if the number of entries in huge-list is 0:
			say "You see nothing unusual in that direction.";
		otherwise:
			say "Looking [D], you can see [huge-list with indefinite articles]."
```

The code above uses a list (a global variable) called the huge-list, which we'll create in the next section. The huge-list is not a permanent thing; we'll assemble it dynamically each time it needs to be used. We'll do this by checking for huge things. A huge thing, as you might imagine, is something that's big enough to see from a distance when outdoors. Nothing in the game will be huge unless the author says it is.

Things that are huge will need to be added to scope in every turn when the player is in an outdoor room. This will assure that the player can refer to them in commands.

```
A thing can be huge. A thing is usually not huge.

Definition: a room is neighboring if the number of moves from it to the location is 1.

After deciding the scope of the player when the location is an outdoor room:
	assemble the huge-list;
	repeat with item running through huge-list:
		place item in scope.

The huge-list is a list of things that varies. The huge-list is {}.

To assemble the huge-list:
	now the huge-list is {};
	let room-inv be a list of things;
	repeat with R running through neighboring rooms:
		now room-inv is {};
		repeat with item running through things in R:
			add item to room-inv;
		repeat with item running through room-inv:
			if item is huge:
				add item to huge-list.
```

```
To assemble the huge-list for (R - a room):
	now the huge-list is {};
	let room-inv be a list of things;
	repeat with item running through things in R:
		add item to room-inv;
	repeat with item running through room-inv:
		if item is huge:
			add item to huge-list.

Instead of examining a huge thing:
	if the noun is not in the location:
		say "[The noun] [is-are] too far away for you to make out any detail.";
	otherwise:
		continue the action.

Before doing anything other than examining with a huge thing:
	if the noun is not enclosed by the location:
		say "[The noun] [is-are] too far away.";
		rule fails;
	otherwise:
		continue the action.
```

Next, we'll implement floorless rooms (more or less in the manner discussed on p. 71). The reason for including floorless rooms in the example will become apparent in a moment.

```
A room can be floorless. A room is usually not floorless. A room has a room called the drop zone. The drop zone of a room is usually nowhere.

After dropping something in a floorless room (called R):
	let DZ be the drop zone of R;
	move the noun to DZ;
	say "[The noun] plummets down out of sight."
```

The final facet of this example is to let the player throw things from one room into another (but only when outdoors). First we'll add a few more vocabulary words as synonyms for “drop”. We really only want these words to be used in the new direction-throwing action, but if we don't add them to the drop action, Inform will respond to the command TOSS THE BALL by saying “What do you want to toss the ball?” This would be ugly.

Note also that if there's any reason in the game why the player won't be allowed to drop things — such as being tied up — the game's code will need to prevent direction-throwing in that circumstance as well. If you omit this step, the player will have an unintended way to drop things, by typing something like THROW RABID GERBIL EAST.

```
Understand "hurl [something]", "toss [something]", and "pitch [something]" as dropping.

Understand "upward" as up.
Understand "downward" as down.
Understand "eastward" as east.
Understand "westward" as west.
Understand "northward" as north.
Understand "southward" as south.

Direction-throwing is an action applying to one thing and one visible thing and requiring light.
Understand "throw [something] [direction]", "hurl [something] [direction]", "pitch [something]
[direction]", and "toss [something] [direction]" as direction-throwing.

Check direction-throwing:
    if the player does not carry the noun:
        say "You can't throw something you're not holding." instead;
    otherwise if the noun is huge:
        say "[The noun] [is-are] too heavy to throw." instead;
    otherwise if the location is not an outdoor room:
        say "[The noun] bounce[s] off the wall and come[s] to rest on the floor.";
        move the noun to the location;
        rule succeeds;
    otherwise:
        let D be the second noun;
        if the room D from the location is nowhere:
            try dropping the noun instead.
```

If the direction-throwing action hasn't been stopped by the check rule, we know (a) that the player is carrying the noun, (b) that the noun is small enough to throw, (c) that the player is outdoors, and (d) that there is a room in that direction, to which the noun can be thrown.

The carry out rule, below, moves the thrown object off into whatever room exists in the direction the player has thrown the object — unless the room in that direction is floorless. If the destination is floorless, the thrown object will instead end up in the drop zone of the floorless room.

The report rule checks to see where the noun has landed. If the player has tried to throw it up into a tree (a floorless room), it will by this time have landed somewhere else. This would most likely be the room the player is in (if she's standing at the base of the tree) — but that's not guaranteed. The player could be out on the limb of a tree and throw something south, toward the main part of the tree, thus causing it to drop to the ground below.

```
Carry out direction-throwing:
    let D be the second noun;
    let R be the room D from the location;
    if R is floorless:
        let DZ be the drop zone of R;
        now the noun is in DZ;
    otherwise:
        now the noun is in R.

Report direction-throwing:
    if the noun is in the location:
        say "You throw [the noun], and [it-they] sail[s] away through the air, but a moment later
[it-they] fall[s] back and land[s] beside you.";
    otherwise:
```

```
		say "You hurl [the noun] [second noun], and [it-they] sail[s] away out of sight."
```

And here’s an entirely pointless scenario with a few rooms and objects, which you can use to test the example. You can try throwing things in various directions while indoors or outdoors, looking various directions while indoors and outdoors, and examining huge things that are far away.

```
The Living Room is an indoor room. "The front door is north, the kitchen south."

The player carries a peach.

The Kitchen is south of the Living Room. The Kitchen is an indoor room. "Not much here. You can go north."

The Yard is an outdoor room. The yard is north of the Living Room. "The lawn wants cutting. The front door is south, and the hill is east."

The Hill is an outdoor room. The Hill is east of the Yard. "From the top of the hill, you can see back down into the yard. There's a tree here you could climb."

The buffalo is in the Hill. The buffalo is huge. The description of the buffalo is "Big and brown."
[Absurdly, the buffalo can be picked up and carried around. This makes it easy to try throwing huge things and so forth.]

The cabbage is in the Hill.

The ferris wheel is in the Hill. The ferris wheel is huge. It is fixed in place. The description of the ferris wheel is "Big and round."

The Treetop is a floorless outdoor room. "You're surrounded by leaves." The Treetop is up from the Hill. The drop zone of the Treetop is the Hill.
```

Appendix C: License

The Inform 7 Handbook is provided for your use under the terms of the Creative Commons Public License (“CCPL” or “License”). This work is protected by copyright and/or other applicable law. Any use of the work other than as authorized under this License or copyright law is prohibited.

By downloading and/or reading this work, you accept and agree to be bound by the terms of this License. To the extent this License may be considered to be a contract, the licensor grants you the rights set forth below in consideration of your acceptance of such terms and conditions.

Creative Commons Public License

1. Definitions

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(g) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. **"Distribute"** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

d. **"License Elements"** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, Noncommercial, ShareAlike.

e. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

f. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

g. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

h. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
i. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.
j. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;
b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";
c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,
d. to Distribute and Publicly Perform Adaptations.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights described in Section 4(e).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(d), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(d), as requested.
b. You may Distribute or Publicly Perform an Adaptation only under: (i) the terms of this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-NonCommercial-ShareAlike 3.0 US) ("Applicable License"). You must include a copy of, or the URI, for Applicable License with every copy of each Adaptation You Distribute or Publicly Perform. You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License. You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.
c. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in con-nection with the exchange of copyrighted works.
d. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, (iv) consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(d) may be implemented in any reasonable manner; provided, however, that in

the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

e. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

   iii. **Voluntary License Schemes**. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c).

f. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING AND TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THIS EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF

LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.
f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.