

## Enumerations

The goal of this handout is to motivate the purpose of enumerations in Java and explain how to create and use them as a part of good Java programming style.

### 1 Motivating Enumerations: Decomposing Monolithic Methods

Two of the most important commandments followed by good programmers are the following:

1. *The Abstraction Commandment*: Thou shalt attempt to capture any repeated computational pattern in a black-box abstraction.
2. *The Modularity Commandment*: Thou shalt compose programs out of reusable mix-and-match parts.

These two commandments go hand-in-hand. A computational pattern that has been captured in a black-box abstraction is typically a good unit of modular reuse. But the “mix-and-match” requirement of modularity implies something more – that the modular parts can be connected via standard kinds of “glue”.

In the real world, excellent examples of standard glue include: power cords and sockets for electrical appliances; nerves transmitting electrochemical signals between the body’s sensors and actuators; standard hardware (e.g., nuts, bolts, screws) for construction; and monetary currency. The currency example is particularly good. Having a common medium of exchange greatly simplifies economic life by removing the need for bartering goods directly. Instead, goods are turned into a common currency, which in turn can be converted into other goods.

As we shall see below, Java’s enumerations are a particularly good kind of glue for connecting program parts together, and thus serve as a kind of common currency in well-written Java programs.

In the domain of programming, many programs can be conceptualized as the composition of instance of common idioms. For instance, a Java method that sums the length of strings in an array of strings conceptually consists of two parts: (1) a *generator* that enumerates the strings of the array one-by-one and (2) an *accumulator* that sums the length of the generated strings. Similarly, a method that determines membership of a given string in a vector of strings consists of two parts: (1) a *generator* that enumerates the strings of the vector one-by-one and (2) an *accumulator* that uses linear search to determine if the given string is among the enumerated ones (returning true immediately as soon as a match is found, but return false if no match is ever found).

In a well-structured program, each idiom should be encapsulated in such a way that it can be reused with instances of other idioms. For instance, it should be possible to combine the strings-from-array generator mentioned in the first example above with the linear search accumulator from the second example above to produce a method that determines if a given string is in an array of strings.

Unfortunately, many programmers often fail to capture such idioms into reusable abstractions. Instead, they tend to manually interweave idioms when writing code, creating what we will call **monolithic** code – code that is a single tightly woven assembly – in contrast with **modular** code that is composed out of mix-and-match reusable parts.

```

import java.util.Vector;

public class Monolithic {
    // Simple monolithic functions that serve to motivate enumerations.

    public static int sumLengthsArray (String [] a) {
        int sum = 0;
        for (int i = 0; i < a.length; i++) {
            sum = sum + a[i].length();
        }
        return sum;
    }

    public static int sumLengthsVector (Vector v) {
        int sum = 0;
        for (int i = 0; i < v.size(); i++) {
            sum = sum + ((String) v.get(i)).length();
        }
        return sum;
    }

    public static boolean memberArray (String s, String [] a) {
        for (int i = 0; i < a.length; i++) {
            if (a[i].equals(s)) {
                return true;
            }
        }
        return false;
    }

    public static boolean memberVector (String s, Vector v) {
        for (int i = 0; i < v.size(); i++) {
            if (v.get(i).equals(s)) {
                return true;
            }
        }
        return false;
    }

    public static void main (String [] args) {
        if (args.length == 0) {
            System.out.println("No arguments to Monolithic");
        } else if (args[0].equals("sumLengthsArray")) {
            System.out.println(sumLengthsArray(ArrayOps.fromString(args[1])));
        } else if (args[0].equals("sumLengthsVector")) {
            System.out.println(sumLengthsVector(StringVector.fromString(args[1])));
        } else if (args[0].equals("memberArray")) {
            System.out.println(memberArray(args[1], ArrayOps.fromString(args[2])));
        } else if (args[0].equals("memberVector")) {
            System.out.println(memberVector(args[1], StringVector.fromString(args[2])));
        } else {
            System.out.println("Unrecognized Monolithic argument: " + args[0]);
        }
    }
}

```

Figure 1: The contents of Monolithic.java.

As a concrete example, consider the `Monolithic` class presented in Fig. 1. (All code presented in this handout can be found in the directory `/home/cs230/download/Enumerations`). This class contains the following four class methods:

- `sumLengthsArray` sums the lengths of the strings in an array of strings.
- `sumLengthsVector` sums the lengths of the strings in a vector of strings.
- `memberArray` determines if a string is a member of an array of strings.
- `memberVector` determines if a string is a member of a vector of strings.

The class also contains a `main` method that facilitates testing these methods. The `main` method uses two auxiliary methods not detailed here – `ArrayOps.fromString` and `VectorOps.fromString` – to turn string representations into arrays of strings and vectors of strings, respectively. Here are some sample uses of the `main` method:

```
$ java Monolithic sumLengthsArray "{Sam,pared,the,yams}"
15
$ java Monolithic memberArray pared "{Sam,pared,the,yams}"
true
$ java Monolithic memberArray pares "{Sam,pared,the,yams}"
false
$ java Monolithic sumLengthsVector "[To,be,or,not,to,be]"
13
$ java Monolithic memberVector be "[To,be,or,not,to,be]"
true
$ java Monolithic memberVector bee "[To,be,or,not,to,be]"
false
```

Note the monolithic nature of the first four methods in Fig. 1. Both `sumLengthsArray` and `memberArray` use an idiom for enumerating through the elements of an array from left to right. However, this idiom is not encapsulated into a single distinct method but is instead hardwired into two different methods. Ditto for the element enumeration idiom embedded in `sumLengthsVector` and `memberVector`. Similarly, `sumLengthsArray` and `sumLengthsVector` illustrate the same accumulation idiom, but this also is not expressed in a separate method. Likewise for the linear search idiom embedded in `memberArray` and `memberVector`.

These examples may seem so simple that it is not necessary to express them as the composition of reusable idioms. But any time an idiom is hardwired into a method, valuable time may be spent remembering details and debugging code. For example, in the array enumeration idiom, should it be `a.length` or `a.length()`? Should the continuation condition use `<` or `<=`? In the vector enumeration idiom, what is the means of determining the number of elements of a vector? In the string length summation idiom, how is the length of a string determined? In an accumulation idiom, under what circumstances is it necessary to downcast the elements of an enumeration? These sorts of questions must be answered again and again when idioms are hardwired into methods, but can be answered once and for all when the idioms are captured into reusable methods.

## 2 Modular Methods Using Enumerations

In Java, an enumeration is an object that can yield elements one at a time, on an as-needed basis. Every enumeration object must have the following two instance methods:

```
public boolean hasMoreElements ();
```

Returns **true** if this enumeration can yield at least one more element, and **false** otherwise.

```
public Object nextElement ();
```

Yields and returns the next element of this enumeration. Calling this method changes the state of the enumeration to prepare to generate the next element after the just-generated one (if there is one). This method should only be called on an enumeration for which `hasMoreElements()` returns **true**; otherwise, invoking `nextElement()` has an unspecified behavior. The fact that the return type of `nextElement()` is `Object` means that downcasts are necessary when using the returned element at a more specific type.

As an example, suppose that `e` names an enumeration for the elements of the string array `{Sam,pared,the,yams}`. Then the following transcript indicates the results of a sequence of statements involving `e`:

```
e.hasMoreElements(); // returns true
e.nextElement(); // returns "Sam"
e.hasMoreElements(); // returns true
e.nextElement(); // returns "pared"
e.hasMoreElements(); // returns true
e.nextElement(); // returns "the"
e.hasMoreElements(); // returns true
e.nextElement(); // returns "yams"
e.hasMoreElements(); // returns false
e.nextElement(); // has an unspecified behavior
```

Because each invocation of `nextElement()` in general returns a different element, it is necessary to extract whatever information is desired from the element returned by `e.nextElement()` (perhaps even storing it somewhere) before invoking `nextElement()` again. For example, the following method encapsulates the linear search accumulation that compares each enumerated element to a given object, return **true** immediately if a match is found:

```
public static boolean member (Object o, Enumeration e) {
    while (e.hasMoreElements()) {
        if (e.nextElement().equals(o)) {
            return true;
        }
    }
    return false;
}
```

In this example, the information extracted from the element returned by `e.nextElement()` is whether or not it is the same string as `o`. Note that the first argument of `member` has the declared type `Object` rather than the more specific `String` type. As we shall see below, this makes the method more reusable.

Because the contract for `nextElement` says that it returns an element of type `Object`, it is necessary to downcast the element when using it at a more specific type. For instance, here is the encapsulation of the string length summation idiom:

```

public static int sumLengths (Enumeration e) {
    int sum = 0;
    while (e.hasMoreElements()) {
        sum = sum + ((String) e.nextElement()).length();
    }
    return sum;
}

```

Note that it is necessary to downcast `e.nextElement()` to type `String` before invoking the string `length` method.

A particularly useful generic accumulator is one that prints all the elements of an enumeration (one per line) and then displays the total number of elements enumerated:

```

public static void test (Enumeration e) {
    int n = 0; // element count
    while (e.hasMoreElements()) {
        System.out.println(e.nextElement());
        n++;
    }
    System.out.println("Total number of elements: " + n);
}

```

Because every object supports the `toString` method implicitly used by `System.out.println`, this accumulator works for *every* enumeration, regardless of the type of its elements. We shall use this method frequently below, where we shall assume that it is declared in the class `EnumTest`. For example, if `e` is an enumeration of the elements in the string array `{Sam,pared,the,yams}`, then `EnumTest.test(e)` displays the following:

```

Sam
pared
the
yams
Total number of elements: 4

```

The above accumulators can be used as components in the modular expression of computational processes. For instance, it turns out that invoking the `elements` method on any `Vector` instance returns an enumeration of all objects in the slots of the vector, from low to high index. Using this, we can express the `sumLengthsVector` and `memberVector` examples from the previous section in a modular way:

```

public static int sumLengthsVector (Vector v) {
    return sumLengths(v.elements());
}

public static boolean memberVector (String s, Vector v) {
    return member(s, v.elements());
}

```

Note how enumerations serve as the glue for connecting the result of the enumeration idiom (`v.elements()`) with the accumulation idioms (`sumLengths` and `member`).

The key to modularity in the above examples is that the accumulation methods work for any enumeration, no matter how it's generated. For example, let's assume that if `array` is an array of strings, then `new StringArrayElts(array)` creates an enumeration of the elements in `array`, from left to right (we'll see how this is defined in the next section). Then we have the following

modular expression of `sumLengthsArray` and `memberArray`.

```
public static int sumLengthsArray (String [] a) {
    return sumLengths(new StringArrayElts(a));
}

public static boolean memberArray (String s, String [] a) {
    return member(s, new StringArrayElts(a));
}
```

We need not stop with enumerations for arrays and strings. For example, if we can create an enumeration of the lines in the file named `filename` (including terminating newlines) using `new FileLines(filename)`, then `sumLengths(new FileLines(filename))` counts the characters in the file. Similarly, if `new FileWords(filename)` enumerates all the words in the file named `filename`, then `member(word, new FileWords(filename))` determines if `word` appears in the file named `filename`.

Because the first argument of `member` has type `Object` rather than `String`, we can even use `member` on enumerations that produce objects other than strings. For example, suppose that `FinArithSeries.between(lo,hi)` generates all the integers (i.e., Java ints wrapped in `Integers`) between `lo` and `hi`, inclusive. Then `member(new Integer(num), FinArithSeries.between(lo,hi))` determines if `num` is between `lo` and `hi`. This is a rather silly example involving integers, but it is easy to imagine applications involving less trivial enumerations involving integers. In fact, we shall see such an example in the next section.

### 3 Defining Enumerations

In the previous sections, we assumed the existence of various enumerations, but did not define any from scratch. Here we show how to define enumeration classes in Java.

#### 3.1 The Java Enumeration Interface

It is important to realize that, in Java, the name `Enumeration` has a technical meaning. It denotes the following entity in the `java.util` package:

```
public interface Enumeration {

    abstract public boolean hasMoreElements();
    abstract public Object nextElement();

}
```

Such an entity is called an **interface**. It is the closest thing that Java provides to a pure specification (contract) with the language itself. The above declaration is a specification of a class that has two public instance methods: a nullary `hasMoreElements` method that returns a `boolean` and a nullary `nextElement` method that returns an `Object`. Any class supplying these two public methods (perhaps along with others) can be explicitly declared to **implement** the `Enumeration` interface.

`Enumeration` can also be used as a type (as in the parameters for the `sumLengths` and `member` methods in the previous section). In this context, it stands for an instance of any class that has been declared as implementing the `Enumeration` interface.

The keyword `abstract` emphasizes that the method headers are purely specification; no code is provide. This stands in contrast to so-called **abstract classes** which can provide some methods

with bodies as well as abstract method headers. Whereas a Java class can have exactly one superclass (from which it inherits variables and methods), it can have any number of interfaces that it implements.

### 3.2 The StringArrayElts Class

As a concrete example of a class implementing the `Enumeration` interface, consider the declaration of the `StringArrayElts` class in Fig. 2. This class encapsulates the step-by-step iteration through the elements of an array of strings. The instance variables `a` (a string array) and `i` (an index into the string array) have been named in a way to make apparent their connection with the variables used in the `for` loops in `sumLengthArray` and `memberArray` methods. Indeed, the state of a `StringArrayElts` instance encapsulates the state of one row of the iteration table corresponding to these `for` loops. From such a state it must be possible to determine if the iteration can continue (i.e., `hasMoreElements` – in this case, is `i` less than the length of the array?) and, if yes, to determine the next row of the iteration table from the current row (update `i` to `i+1`).

Actually, the update is a little bit tricky. The variable `i` must be incremented before an invocation of `nextElement` returns, but then returning `a[i]` would exit the `nextElement` method with the wrong element. There are many ways to address this. The approach show in Fig. 2 is to perform the increment but then return `a[i-1]`. Here are two of many possible alternatives:

1. The result to be returned (`a[i]` before the increment to `i`) can be saved in a temporary variable before the increment and returned later:

```
public Object nextElement () {
    int result = a[i]
    i = i+1;
    return result;
}
```

2. We can take advantage of the subtle (and easy to get wrong!) detail that `i++` is a so-called **post-increment** operator that, in addition to incrementing `i` returns the value of `i` *before* the increment. It is called a **post-increment** operator because it the increment is effectively performed on the number *after* it is returned:

```
public Object nextElement () {
    return a[i++];
}
```

A few other things to note in the `StringArrayElts` example are:

- Because `Enumeration` (like `Vector`) is in the `java.util` package rather than the automatically imported `java.lang` package, it is necessary to begin the file with the explicit import declaration

```
import java.util.Enumeration;
```

To import all entities from `java.util` (including `Vector` as well as `Enumeration`), use the following declaration:

```
import java.util.*;
```

- The phrase `implements Enumeration` in the declaration of `StringArrayElts` is necessary in order for instances of `StringArrayElts` to be used as objects matching type `Enumeration`.

```
import java.util.Enumeration;

public class StringArrayElts implements Enumeration {

    // Instance variables
    private String [] a;
    private int i;

    // Constructor method
    public StringArrayElts (String [] a) {
        this.a = a;
        i = 0;
    }

    // Instance methods required for enumerations
    public boolean hasMoreElements () {
        return (i < a.length);
    }

    public Object nextElement () {
        i = i+1;
        return a[i-1];
    }

    // Handy testing method
    public static void main (String [] args) {
        EnumTest.test(new StringArrayElts(ArrayOps.fromString(args[0])));
    }
}
```

Figure 2: StringArrayElts enumerates the elements of a string array.

- The `main` method of the `StringArrayElts` class is used to perform a test on a simple use of `StringArrayElts`. For example:

```
$ java StringArrayElts "{Sam,pared,the,yams}"
Sam
pared
the
yams
Total number of elements: 4
```

### 3.3 The VectorElts Class

As mentioned above, enumerations of vectors can be created by invoking the nullary `elements` method on a `Vector` instance. However, for pedagogical purposes, in Fig. 3 we present the details of a `Vector` enumeration class that we have “rolled on our own”. This is very similar to the `StringArrayElts` class, except:

- It must import `java.util.Vector` in addition to `java.util.Enumeration`;
- The state variable `v` holds a `Vector` rather than an array;
- `v.size()` returns the number of elements in the vector;
- `v.get(i)` returns the element in the vector slot numbered `i`;
- `StringVector.fromString` is used to parse a string representation of a vector into a `Vector` instance.

### 3.4 The StringChars Class

Yet another class similar in construction to `StringArrayElts` and `VectorElts` is `StringChars`, which enumerates the characters from a string (Fig. 4). A few things to note:

- The `len` variable remembers the length of the string, so that `s.length()` does not need to be invoked for every invocation of `hasMoreElements` and `nextElement`.
- Because `char` is not an `Object` instance, it is necessary to wrap each character in an instance of the `Character` class. (It can be unwrapped via the invocation `charValue()`).
- Unlike `StringArrayElts` and `VectorElts`, the `hasMoreElements` method in `StringChars` explicitly handles the case of attempting to call `nextElement` when there are no more characters to enumerate. It does this by signalling an instance of the `RuntimeException` class, which is accomplished in Java via the `throw` construct. In `StringArrayElts` and `VectorElts`, a similar situation would signal a index out-of-bounds exception. The explicit exception in `StringChars` gives a more informative error message.

```

import java.util.Enumeration;
import java.util.Vector;

public class VectorElts implements Enumeration {

    // Instance variables
    private Vector v;
    private int i;

    // Constructor method
    public VectorElts (Vector v) {
        this.v = v;
        i = 0;
    }

    // Instance methods required for enumerations
    public boolean hasMoreElements () {
        return (i < v.size());
    }

    public Object nextElement () {
        i = i+1;
        return v.get(i-1);
        // Alternatively, replace the above two lines by:
        // return v.get(i++);
    }

    // Handy testing method
    public static void main (String [] args) {
        EnumTest.test(new VectorElts(StringVector.fromString(args[0])));
    }
}

```

Figure 3: VectorElts enumerates the elements of a vector.

```

import java.util.Enumeration;

public class StringChars implements Enumeration {
    // Class that enumerates characters from a string.

    // Instance variables
    String s; // the string
    int len; // the length of the string;
    int i; // the current index into the string

    // Constructor method
    public StringChars (String str) {
        s = str;
        len = str.length();
        i = 0;
    }

    // Instance methods implementing Enumeration interface
    public boolean hasMoreElements() {
        return (i < len);
    }

    public Object nextElement() {
        if (i < len) {
            char c = s.charAt(i);
            i++;
            return new Character(c);
        } else {
            throw new RuntimeException("StringChars: no more elements!");
        }
    }

    // Static method for testing this class
    public static void main (String [] args) {
        EnumTest.test(new StringChars(args[0]));
    }
}

```

Figure 4: StringChars enumerates the characters of a string.

### 3.5 The FileChars Class

As a more complex example of an enumeration, consider `FileChars`, which enumerates the characters in a file (Figs. 5–6). This enumeration has an instance variable `rdr` for an instance of Java's `FileReader` class, which is used to read the characters in a file one-by-one. `FileReader` instances are created by invoking the constructor method `FileRead` on the name of the file. Invoking the `read()` method on an instance of `FileReader` yields the the ASCII integer value of the next character of the file. The reason that `read()` returns an `int` rather than a `char` is that the distinguished integer `-1` is used to indicate that there are no more characters (i.e., the end of the file has been reached).

```
import java.io.*;
import java.util.Enumeration;

public class FileChars implements Enumeration {
    // Class that enumerates characters from a file.

    // Instance variables
    Reader rdr;
    int buff; // buffer holding "peek" character, if any; -1 for no character.

    // Constructor method
    public FileChars (String filename) {
        try {
            rdr = new FileReader(filename);
            buff = -1;
        } catch (FileNotFoundException e) {
            throw new RuntimeException("File not found: " + filename);
        }
    }

    // Instance methods implementing Enumeration interface
    public boolean hasMoreElements() {
        int i = peekChar(); // look at, but don't remove, next char;
        return (i != -1);
    }

    public Object nextElement() {
        int i = getChar(); // look at and remove next char;
        if (i == -1) {
            throw new RuntimeException("FileChars: no more elements!");
        } else {
            return new Character((char) i);
        }
    }
}
```

Figure 5: Implementation of the file character enumeration, `FileChars` (Part 1).

The implementation of `FileChars` illustrates an important **buffering idiom** often encountered when implementing enumerations. In order to determine if there are more characters, `hasMoreElements` needs to invoke `rdr.read()` to see if it returns `-1`. But if it doesn't return `-1`, this invocation yields the next character in the file, which needs to be stored somewhere so that the next call to `nextElement` can return it. For this purpose, a one-character buffer named `buff`

```

// Auxiliary instance methods

public int getChar() {
    // Returns and removes next character (as int).
    // Returns -1 for an empty file.
    if (buff != -1) {
        int result = buff;
        buff = -1; // empty the buffer
        return result;
    } else {
        try {
            return rdr.read();
        } catch (IOException e) {
            throw new RuntimeException ("FileChars.getChar(): "
                + e.getMessage());
        }
    }
}

public int peekChar() {
    // Returns, but does not remove, next character (as int).
    // Returns -1 for an empty file.
    if (buff != -1) {
        return buff;
    } else {
        try {
            buff = rdr.read();
            return buff;
        } catch (IOException e) {
            throw new RuntimeException ("FileChars.peekChar(): "
                + e.getMessage());
        }
    }
}

// Static method for testing this class
public static void main (String [] args) {
    EnumTest.test(new FileChars(args[0]));
}
}

```

Figure 6: Implementation of the file character enumeration, FileChars (Part 2).

is used. Since it is necessary to distinguish an empty buffer (which holds no character) from a full one (which stores a single character), the `buff` variable actually has `int` type rather than `char` type, which allows `-1` to indicate an empty buffer.

The buffer is used in the following way by the two enumeration methods:

- **hasMoreElements**: If `buff` is full, `hasMoreElements` returns `true` since there is at least one more character (the contents of the buffer) to enumerate. Otherwise, it is necessary to “peek” at the next character in the file, if there is one, and store it into `buff` so it is not forgotten.
- **nextElement**: If `buff` contains a character, `nextElement` returns it, emptying `buff` by setting it to `-1` before returning. Otherwise, `nextElement` gets the next character from the file (if there is one) and returns it, maintainin an empty buffer in this case.

The auxiliary methods `peekChar` and `getChar` abstract over buffer manipulation details. The `peekChar` method returns the next character in the buffer or file, but does not consume it. In contrast, `getChar` consumes and returns the next character in the buffer or file.

Buffers like `buff` are commonly needed in enumeration implementations in cases where an action performed by `hasMoreElements` has the side effect of potentially generating one or more elements. In this case, the generated elements must be stored in a buffer so that they are not lost.

A few other details of the `FileChars` implementation are worth noting:

- The declaration `import java.io.*;` makes Java’s IO (i.e., Input/Output) methods visible to the compiler.
- It turns out that certain Java invocations, such as `new FileReader(filename)` and `rdr.read()`, may raise so-called IO exceptions, and Java *requires* that these exceptions are handled or else the compiler will issue a compile-time error. Exception handling in Java is performed via the construct

```
try {stm1} catch (exception-type e) {stm2}.
```

The `try/catch` construct first executes `stm1`. If the execution of `stm1` returns normally, then the `try/catch` statement is done, and `stm2` is never executed. But if an exception is raised during the execution of `stm1` and the type of that exception matches `exception-type`, then `stm2` is executed.

- As in `StringChars`, it is necessary to wrap the characters enumerated by `FileChars` in an instance of the `Character` class.
- An alternative to using a single `int` variable for the character buffer would be to use a pair of variables: a `char` variable to hold the buffered character (if there is one) and an `boolean` variable to indicate if the buffer is full (`true`) or empty (`false`).

As an example of `FileChars` in action, suppose that `animals.txt` is a file containing the following three lines:

```
bat
cat
dog
```

Then invoking the `main` method of `FileChars` on `animals.txt` has the following behavior:

```
$ java FileChars animals.txt
b
a
t

c
a
t

d
o
g
```

Total number of elements: 12

The reason that there are *two* blank lines after the last character of each line is that displaying the newline character at the end of a line adds an extra line.

### 3.6 The FileLines Class

As a second example of the buffering idiom, consider the implementation of a `FileLines` class that enumerates the lines of a file one by one (Fig. 7). Like `FileChars`, `FileLines` has a `rdr` instance variable holding an instance of a `FileReader` for enumerating the characters of the file one-by-one. However, since `FileLines` enumerates strings rather than characters, it needs some place to store all the characters of the current line until the newline is encountered. The `StringBuffer` variable `sb` serves the role of buffer in `FileLines`.

Note that `hasMoreElements` only returns `false` when the string buffer is empty *and* there are no more characters in the file.

Also note that the enumerated lines contain the terminating newline character. It would be easy to omit this if it were not desired.

### 3.7 Filtering Enumerations: The StringLengthFilter Class

It is often desirable to filter an enumeration – to allow some of the elements to pass through while blocking others. For example, given an enumeration of strings, we might want to pass only those strings with a certain minimal length.

As an example of the filtering idiom, consider the `StringLengthFilter` class in Fig. 8. If `n` is an integer and `e` is an `Enumeration` of strings, then `new StringLengthFilter(n, e)` is an enumeration that has all the strings of `e` with length greater than or equal to `n`, in the same order of appearance as in `e`.

Like `FileChars` and `FileLines`, `StringLengthFilter` must maintain a buffer (the variable `buff` of type `String`). The reason is that the `hasMoreElements` method of `StringLengthFilter` must generate the strings of the enumeration stored in the instance variable `enum` until one is found with length  $\geq \text{len}$ . If there is such a string, it is stored in `buff` so that the next call to `nextElement` can return it. If there is no such string, `buff` remains empty (i.e., contains the `null` pointer) and `hasMoreElements` returns `false`.

The `peekString` and `getString` auxiliary methods are abstractions for manipulating the string buffer. They are similar to the `peekChar` and `getChar` methods of `FileChars`. Note the downcast

```

import java.io.*;
import java.util.Enumeration;

public class FileLines implements Enumeration {
    // Enumerates lines of a file (including terminating newlines, if any)

    // Instance variables
    Reader rdr;
    StringBuffer sb;

    // Constructor method
    public FileLines (String filename) {
        try { rdr = new FileReader(filename);
            sb = new StringBuffer();
        } catch (FileNotFoundException e) {
            throw new RuntimeException("File not found: " + filename);
        }
    }

    // Instance methods implementing Enumeration interface
    public boolean hasMoreElements() {
        try { if (sb.length() > 0) {
            return true;
        } else {
            int i = rdr.read();
            if (i == -1) { // EOF encountered
                return false;
            } else {
                sb.append((char) i);
                return true;
            }
        }
    } catch (IOException e) { throw new RuntimeException("IOException: " + e.getMessage()); }
}

    public Object nextElement() {
        try { while (! isNewline(sb.charAt(sb.length()-1))) {
            int i = rdr.read();
            if (i == -1) { // EOF encountered
                return sb.toString();
            } else {
                sb.append((char) i);
            }
        }
        String s = sb.toString();
        sb.setLength(0);
        return s;
    } catch (IOException e) { throw new RuntimeException("IOException: " + e.getMessage()); }
}

    // Auxiliary instance method
    private static boolean isNewline (char c) { return (c == '\n'); }

    // Static method for testing this class
    public static void main (String [] args) { EnumTest.test(new FileLines(args[0]));*s }
}

```

Figure 7: Implementation of the file lines enumeration, FileLines.

```

import java.util.Enumeration;

public class StringLengthFilter implements Enumeration {
    // A filter that passes only strings whose length is >= len.

    // Instance variables
    private Enumeration enum;
    private int len;
    private String buff;

    // Constructor method
    public StringLengthFilter (int len, Enumeration enum) {
        this.enum = enum;
        this.len = len;
        buff = null;
    }

    // Instance methods for Enumeration
    public boolean hasMoreElements() { peekString(); return (buff != null); }

    public Object nextElement() { return getString(); } // returns null when no more elements.

    // Auxiliary class methods
    public String peekString() { // Return (but don't consume) next string with length >= len.
        while ((buff == null) && enum.hasMoreElements()) {
            String s = (String) enum.nextElement();
            if (s.length() >= len) {
                buff = s;
            }
        } // Invariant: (buff != null) || (! enum.hasMoreElements())
        return buff;
    }

    public String getString() { // Return and consume the next string with length >= len.
        peekString();
        String result = buff; // may be null!
        buff = null;
        return result;
    }

    // Handy testing method
    public static void main (String [] args) {
        try { if (args[0].equals("StringArrayElt")) {
            EnumTest.test(new StringLengthFilter(
                Integer.parseInt(args[1]),
                new StringArrayElt(ArrayOps.fromString(args[2]))));
        } else if (args[0].equals("FileLines")) {
            EnumTest.test(new StringLengthFilter(
                Integer.parseInt(args[1]),
                new FileLines(args[2])));
        }
    } catch (NumberFormatException e) {
        throw new RuntimeException ("NumberFormatException: " + e.getMessage());
    }
}

```

Figure 8: Implementation of the string length filter, `StringLengthFilter`.

to `String` performed within `peekChar`; this is necessary since the `String` length method is invoked on the enumerated element.

Here is an example of `StringArrayElts` in action:

```
$ java StringLengthFilter StringArrayElts 4 "{Sam,pared,the,yams}"
pared
yams
Total number of elements: 2
$ java StringLengthFilter StringArrayElts 5 "{Sam,pared,the,yams}"
pared
Total number of elements: 1
$ java StringLengthFilter StringArrayElts 6 "{Sam,pared,the,yams}"
Total number of elements: 0
```

Because `StringFilterLength` is a modular component, it can easily be composed with other components. For example, here is a method for summing the lengths of all strings in an array whose length is greater than a given threshold:

```
public static int sumLengthsArrayThreshold (int threshold, String [] a)
    return sumLengths(new StringLengthFilter(threshold, new StringArrayElts(a)));
```

It is possible to instantiate the filtering idiom represented by `StringLengthFilter` for any filtering predicate. According to the first commandment given at the beginning of this document, we should attempt to capture the filtering pattern in a separate method that can be instantiated for different predicates. It is possible to do this, but the details in Java are messy, and we choose not to show them here.

### 3.8 Mapping Enumerations: The `MapLowerCase` Class

Another common idiom is the **mapping** idiom, in which some function is performed to each element of an enumeration. We illustrate the mapping idiom via the `MapLowerCase` class (Fig. 9), which maps each string in an enumeration to its lowercase form.

As is apparent from the straightforward implementation of `MapLowerCase`, mapping is much easier to implement than filtering. There is no need for any buffering, since the number of elements generated by the mapper is exactly the same as the number of elements generated by the `Enumeration` in its instance variable `enum`.

Here is an example of `MapLowerCase` in action:

```
$ java MapLowerCase StringArrayElts "Neither,HERE,nor,THERE"
neither
here
nor
there
Total number of elements: 4
```

Of course, `MapLowerCase` can be combined with other components. For example,

```
new MapLowerCase(
    new StringLengthFilter(5, new StringArrayElts
        (ArrayOps.fromString "Neither,HERE,nor,THERE")))
```

denotes an `Enumeration` of the two strings `"neither"` and `"there"`.

```

import java.util.Enumeration;

public class MapLowerCase implements Enumeration {
    // A mapper that converts every string to lower case.

    // Instance variables
    private Enumeration enum;

    // Constructor method
    public MapLowerCase (Enumeration enum) {
        this.enum = enum;
    }

    // Instance methods for Enumeration
    public boolean hasMoreElements() {
        return enum.hasMoreElements();
    }

    public Object nextElement() {
        return ((String) enum.nextElement()).toLowerCase();
    }

    // Handy testing method
    public static void main (String [] args) {
        if (args[0].equals("StringArrayElts")) {
            EnumTest.test(new MapLowerCase(
                new StringArrayElts(ArrayOps.fromString(args[1]))));
        } else if (args[0].equals("FileLines")) {
            EnumTest.test(new MapLowerCase(new FileLines(args[1])));
        }
    }
}

```

Figure 9: Implementation of the lowercase mapper, MapLowerCase.

### 3.9 Integer Enumerations: The `FinArithSeries` Class

It is possible to enumerate integers as well as strings and characters. Of course, because the `Enumeration` interface requires `nextElement` to return an `Object`, an integer enumeration must package up all Java `ints` into instances of the `Integer` wrapper class.

As an example of integer enumerations, consider the `FinArithSeries` class in Figs. 10–11. This class generates finite arithmetic series – i.e., sequences of integers in which all consecutive pairs differ by the same integer. Here are some examples of the `FinArithSeries` class in action:

```
$ java FinArithSeries between 3 7
3
4
5
6
7
Total number of elements: 5

$ java FinArithSeries between 8 5
8
7
6
5
Total number of elements: 4

$ java FinArithSeries betweenStep 4 20 3
4
7
10
13
16
19
Total number of elements: 6

$ java FinArithSeries upTo 5
1
2
3
4
5
Total number of elements: 5

$ java FinArithSeries downFrom 5
5
4
3
2
1
Total number of elements: 5
```

An instance of `FinArithSeries` has three instance variables: the next integer to be generated (`i`), the (inclusive) limit, either upper or lower, for the series (`limit`), and the difference between consecutive pairs (`step`). If `step` is non-negative, `limit` is interpreted as an upper limit, while if `step` is negative, `limit` is treated as a lower limit.

The `hasMoreElements` and `nextElement` methods are straightforward. The class methods `betweenStep`, `between`, `upTo` and `downFrom` invoke the constructor method for `FinArithSeries`

```

import java.util.Enumeration;

public class FinArithSeries implements Enumeration {
    // Class for enumerating finite arithmetic series of integers

    // Instance variables
    private int i;
    private int limit;
    private int step;

    // Constructor method
    public FinArithSeries(int i, int limit, int step) {
        this.i = i;
        this.limit = limit;
        this.step = step;
    }

    // Instance methods required by enumerations
    public boolean hasMoreElements () {
        return ((step >= 0) && (i <= limit))
            || ((step < 0) && (i >= limit));
    }

    public Object nextElement () {
        i = i + step;
        return new Integer(i - step);
    }

    // Class methods for creating different flavors of series
    public static FinArithSeries betweenStep (int start, int stop, int step) {
        return new FinArithSeries(start, stop, step);
    }

    public static FinArithSeries between (int start, int stop) {
        if (stop >= start) {
            return betweenStep(start, stop, 1);
        } else {
            return betweenStep(start, stop, -1);
        }
    }

    public static FinArithSeries upTo (int hi) {
        if (hi >= 1) {
            return between(1, hi);
        } else {
            throw new RuntimeException ("upTo: non-positive limit" + hi);
        }
    }

    public static FinArithSeries downFrom (int hi) {
        if (hi >= 1) {
            return between(hi, 1);
        } else {
            throw new RuntimeException ("downFrom: non-positive start" + hi);
        }
    }
}

```

Figure 10: Implementation of FinArithSeries (Part 1)

```

// Handy testing method
public static void main (String [] args)          try {
    if (args.length == 0) {
        System.out.println("No arguments to FinArithSeries");
    } else if (args[0].equals("betweenStep")) {
        EnumTest.test(betweenStep(Integer.parseInt(args[1]),
            Integer.parseInt(args[2]),
            Integer.parseInt(args[3])));
    } else if (args[0].equals("between")) {
        EnumTest.test(between(Integer.parseInt(args[1]),
            Integer.parseInt(args[2])));
    } else if (args[0].equals("upTo")) {
        EnumTest.test(upTo(Integer.parseInt(args[1])));
    } else if (args[0].equals("downFrom")) {
        EnumTest.test(downFrom(Integer.parseInt(args[1])));
    } else {
        System.out.println("Unrecognized FinArithSeries method: " + args[0]);
    }
} catch (NumberFormatException e) {
    throw new RuntimeException ("NumberFormatException: " + e.getMessage());
}
}
}

```

Figure 11: Implementation of `FinArithSeries` (Part 2)

with appropriate arguments.

We can use integer enumerations with other components. For example, here is a summation method that sums all of the integers in an integer enumeration:

```

public static int sum (Enumeration e) {
    int sum = 0;
    while (e.hasMoreElements()) {
        sum = sum + ((Integer) e.nextElement()).intValue();
    }
    return sum;
}

```

Using the sum accumulator, we can find the sum of arithmetic series. For instance, to find the sum of the integers from 1 to 100, we can evaluate the expression `sum (FinArithSeries.upTo(100))`.

### 3.10 Infinite Enumerations

Thus far, all enumerations we have considered have been finite. That is, if we keep calling `nextElement()`, eventually `hasMoreElements()` will return `false`. But it is also possible to have *infinite* enumerations – enumerations that will never stop generating elements.

As a simple example of an infinite enumeration, consider the `InfArithSeries` class in Fig. 12. This is similar to `FinArithSeries`, except that there is no `limit` instance variable, and `hasMoreElements` always returns `true`. For instance, `InfArithSeries.upFrom(17)` yields the infinite series of integers that begins 17, 18, 19, 20, ..., and `InfArithSeries.upFromBy(5, 3)` yields the infinite series of integers that begins 5, 8, 11, 14, .... Of course, if you invoke one of these methods via the main method, the invocation will not terminate – you'll have to terminate it yourself by typing Control-C.

As another example of an infinite enumeration, consider the `Fibonacci` class in Fig. 13. Recall that the first two Fibonacci numbers are 0 and 1, and each subsequent number is the sum of the two

```

public class InfArithSeries implements Enumeration {
    // Class for enumerating infinite arithmetic series of integers

    // Instance variables
    private int i;
    private int step;

    // Constructor methods
    public InfArithSeries(int i, int step) {
        this.i = i;
        this.step = step;
    }

    // Instance methods required by enumerations
    public boolean hasMoreElements () { return true; }

    public Object nextElement () {
        i = i + step;
        return new Integer(i - step);
    }

    public static InfArithSeries upFrom (int start) {
        return new InfArithSeries(start, 1); }

    public static InfArithSeries upFromBy (int start, int step) {
        return new InfArithSeries(start, step); }

    public static InfArithSeries downFrom (int start) {
        return new InfArithSeries(start, -1); }

    public static InfArithSeries downFromBy (int start, int step) {
        return new InfArithSeries(start, -step); }

    // Handy testing method
    public static void main (String [] args) {
        try {
            if (args.length == 0) {
                System.out.println("No arguments to IntElts");
            } else if (args[0].equals("upFrom")) {
                EnumTest.test(upFrom(Integer.parseInt(args[1])));
            } else if (args[0].equals("upFromBy")) {
                EnumTest.test(upFromBy(Integer.parseInt(args[1]),
                    Integer.parseInt(args[2])));
            } else if (args[0].equals("downFrom")) {
                EnumTest.test(downFrom(Integer.parseInt(args[1])));
            } else if (args[0].equals("downFromBy")) {
                EnumTest.test(downFromBy(Integer.parseInt(args[1]),
                    Integer.parseInt(args[2])));
            } else {
                System.out.println("Unrecognized InfArithSeries method: " + args[0]);
            }
        } catch (NumberFormatException e) {
            throw new RuntimeException ("NumberFormatException: " + e.getMessage());
        }
    }
}

```

Figure 12: Implementation of the InfArithSeries class

previous numbers. So the infinite sequence of Fibonacci numbers begins 0, 1, 1, 2, 3, 5, 8, 13, 21, . . . . The `Fibonacci` class generates these numbers using two instance variables `a` and `b` that stand for two consecutive numbers in the Fibonacci sequence. These are initialized to 0 and 1 respectively, and at each call to `nextElement`, the values are updated in such a way to “shift” the sequence by one position. There is also a two-argument constructor method that allows the creation of Fibonacci-like sequences for initial values other than 0 and 1.

```
import java.util.Enumeration;

public class Fibonacci implements Enumeration {

    private int a;
    private int b;

    public Fibonacci () {
        this(0,1);
    }

    public Fibonacci (int a, int b) {
        this.a = a;
        this.b = b;
    }

    public boolean hasMoreElements () {
        return true;
    }

    public Object nextElement () {
        int result = a;
        a = b;
        b = b + result;
        return new Integer(result);
    }

    // Handy testing method
    public static void main (String [] args) {
        try {
            if (args.length == 0) {
                EnumTest.test(new Fibonacci());
            } else if (args.length == 2) {
                EnumTest.test(new Fibonacci(Integer.parseInt(args[0]),
                                             Integer.parseInt(args[1])));
            } else {
                System.out.println("Unrecognized Fibonacci method");
            }
        } catch (NumberFormatException e) {
            throw new RuntimeException ("NumberFormatException: " + e.getMessage());
        }
    }
}
```

Figure 13: Implementation of the `Fibonacci` class

An example of the initial values enumerated by `new Fibonacci()` is shown in Fig. 14. Note that there are some negative numbers at the end of the transcript. How can this be? This is an artifact of Java’s finite integer representations. The Java `int` type has a “largest” integer value, and counting above this value “wraps around” to a negative value with large magnitude.

```
$ java Fibonacci
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
267914296
433494437
701408733
1134903170
1836311903
-1323752223
512559680
-811192543
```

Figure 14: Initial numbers enumerated by the enumeration `new Fibonacci()`.

What can be done with an infinite enumeration? Although it's obviously impossible to process every element of an infinite enumeration, it *is* possible to process a finite prefix of an infinite enumeration. Many computations involve finite prefixes of an infinite enumeration, where the length of the prefix is not necessarily known in advance.

As a simple example of finite prefixes, consider the `Prefix` class in Fig. 15, which allows truncating a given enumeration after a specified number of elements has been enumerated. For example, `new Prefix(10, new Fibonacci())` denotes an enumeration of the first 10 Fibonacci numbers (0, 1, 1, 2, 3, 5, 8, 13, 21, 34) and `new Prefix(7, InfArithSeries.upFromBy(17,3))` denotes the enumeration of the seven numbers 17, 20, 23, 26, 29, 32, 35. These finite enumerations can be processed like any other finite enumerations. For instance, we can sum the first 10 Fibonacci numbers via the following expression:

```
sum(new Prefix(10, new Fibonacci())).
```

```

import java.util.Enumeration;

public class Prefix implements Enumeration {

    // Instance variables
    private int n; // The number of elements to be enumerated
    private Enumeration enum;

    public Prefix (int n, Enumeration enum) {
        this.n = n;
        this.enum = enum;
    }

    public boolean hasMoreElements () {
        return (enum.hasMoreElements()) && (n > 0);
    }

    public Object nextElement () {
        Object next = enum.nextElement();
        n--; // decrement the number of elements to be enumerated.
        return next;
    }

    public static int sum (Enumeration e) {
        int sum = 0;
        while (e.hasMoreElements()) {
            sum = sum + ((Integer) e.nextElement()).intValue();
        }
        return sum;
    }

    // Handy testing function
    public static void main (String [] args) {
        try {
            if (args[1].equals("InfArithSeries.upFrom")) {
                EnumTest.test(new Prefix(Integer.parseInt(args[0]),
                    InfArithSeries.upFrom(Integer.parseInt(args[2]))));
            } else if (args[1].equals("InfArithSeries.upFromBy")) {
                EnumTest.test(new Prefix(Integer.parseInt(args[0]),
                    InfArithSeries.upFromBy(Integer.parseInt(args[2]),
                    Integer.parseInt(args[3]))));
            } else if (args[1].equals("Fibonacci")) {
                EnumTest.test(new Prefix(Integer.parseInt(args[0]), new Fibonacci()));
            } else {
                throw new RuntimeException("Unrecognized option: Prefix");
            }
        } catch (NumberFormatException e) {
            throw new RuntimeException ("NumberFormatException: " + e.getMessage());
        }
    }
}

```

Figure 15: Implementation of the Prefix class