Inheritance

The Power of OO Programming!

Geometric Shapes

- The “canonical” example, introduced by Alan Kay
- You’ll see it in Lab!
Inheritance

- Inheritance is a fundamental technique used to
- The child is a more specific version of parent
- The child inherits characteristics of the parent (methods and data defined by the parent class)
- Tailor derived class by adding new variables or methods, or by modifying the inherited ones
- Keyword extends is used to establish an inheritance (is-a) relationship

```java
class Child extends Parent {
    // class contents
}
```

Class Hierarchies

- A child class can be the parent of another child, forming a class hierarchy
- Two children of the same parent are called siblings
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
  - Therefore, a child class inherits from all its ancestor classes
```java
/** Words.java
 * Demonstrates the use of an inherited method.
 * @author Java Foundations
 */

public class Words {
    /**
     * Driver: Instantiates a derived class and invokes its inherited
     * and local methods.
     */
    public static void main(String[] args) {
        Dictionary webster = new Dictionary();
        System.out.println("Number of pages: " + webster.getPages());
        System.out.println("Number of definitions: " + webster.getDefinitions());
        System.out.println("Definitions per page: " + webster.computeRatio());
    }
}
```

```java
/**
 * Represents a book. Used as the parent of a derived class to
 * demonstrate inheritance and the use of the super reference.
 * @author Java Foundations
 */

public class Book {
    protected int pages = 1500;

    // Note: No constructor

    /**
     * Pages setter (mutator)
     * @param numPages number of pages in book
     */
    public void setPages(int numPages) {
        pages = numPages;
    }

    /**
     * Pages getter (accessor)
     * @return number of pages in book
     */
    public int getPages() {
        return pages;
    }
}
```
The **protected** Modifier

- A **protected** variable is visible to any class in the same package as the parent class.

- The **protected** modifier allows a child class to reference a variable or method directly in the parent class.

- It provides more **encapsulation** than public visibility, but is not as tightly encapsulated as private visibility.
The **super** Reference

- Constructors are **not** inherited, even though they have public visibility
  - Yet, we often want to use the parent’s constructor to set up the “parent’s part” of the object
- The keyword **super** can be used to refer to the parent class, including the parent’s constructor
- A child’s constructor should:
  - Call the parent’s constructor as it’s first line: `super();`
  - If it does not call `super()`, a 0-parameters `super()` constructor will be called anyway!
- The **super** reference can also be used to reference other variables & methods defined in parent’s class

---

**Words2.java**

```java
/**
 * Demonstrates the use of an inherited method.
 * Author Java Foundations
 */
public class Words2 {
    /**
     * Driver: Instantiates a derived class and invokes its inherited
     * and local methods.
     */
    public static void main (String[] args) {
        Dictionary2 webster = new Dictionary2(1500, 52500);
        System.out.println ("Number of pages: " + webster.getPages());
        System.out.println ("Number of definitions: " + webster.getDefinitions());
        System.out.println ("Definitions per page: " + webster.computeRatio());
    }
}
```
```java
/**
* Represents a book. Used as the parent of a derived class to
* demonstrate inheritance and the use of the super reference.
* @author Java Foundations
*/
public class Book2 {
    protected int pages;

    // Constructor (will be invoked by child)
    public Book2 (int numPages) {
        pages = numPages;
    }

    /**
     * @param numPages number of pages in the book
     */
    public void setPage(int numPages) {
        pages = numPages;
    }

    /**
     * @return number of pages in the book
     */
    public int getPage() {
        return pages;
    }
}
```

```java
/**
* Represents a dictionary, which is a book. Used to demonstrate
* the use of the super reference.
* @author Java Foundations
*/
public class Dictionary2 extends Book2 {
    private int definitions;

    /**
     * Constructor. Note: Calls super
     */
    public Dictionary2(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }

    public double computeRatio() {
        return definitions/pages;
    }

    public void setDefinitions(int numDefinitions) {
        definitions = numDefinitions;
    }

    public int getDefinitions() {
        return definitions;
    }
}
```
Overriding

* What happens when a parent and a child class have methods with the same name?

* A child class can override the definition of an inherited method in favor of its own

* A method in the parent class can be invoked explicitly using the super reference, as in:

```java
super.message()
```

Thought.java

```java
/**
 * Represents a stray thought. Used as the parent of a derived class to demonstrate the use of an overridden method.
 * @author Java Foundations
 */
public class Thought{
    /**
     * Prints a message
     */
    public void message(){
        System.out.println("I feel like I'm diagonally parked in a " + "parallel universe.");
    }
}
```
```java
/**
 * Represents some thoughtful advice. Used to demonstrate the
 * use of an overridden method.
 * @author Java Foundations
 */

public class Advice extends Thought {
  /**
   * Prints a message. Overrides the parent's version.
   */
  public void message(){
    System.out.println("Warning: Dates in calendar are closer " +
        "than they appear.");
    //explicitly invokes the parent's version
    super.message();
  }
}
```

```java
/**
 * Demonstrates the use of an overridden method.
 * @author Java Foundations
 */

public class Messages {
  /**
   * Creates two objects and invokes the message method in each
   */
  public static void main(String[] args) {
    Thought parked = new Thought();
    Advice dates = new Advice();
    parked.message();
    dates.message(); //overridden
  }
}
```
Overriding

- A child class can *override* the definition of an inherited method in favor of its own
- A method in the parent class can be invoked explicitly using the `super` reference, as in:
  ```java
  super.message()
  ```
- If a method is declared with the `final` modifier, it *cannot* be overridden
- The concept of overriding can be applied to variables and is called *shadowing variables*. Shadowing variables should be **avoided** because it tends to cause unnecessarily confusing code

Overloading vs. Overriding

- **Overloading** deals with multiple methods with the same name in the *same class*, but with *different signatures*
- **Overriding** deals with two methods, one in a *parent class* and one in a *child class*, that have the *same signature*
- Overloading lets you define a similar operation in different ways for different parameters
- Overriding lets you define a similar operation in different ways for different object types
The Object Class

The mother of all classes!

- A class called Object is defined in the java.lang package of the Java standard class library
- All classes are derived from the Object class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the Object class
- Therefore, the Object class is the ultimate root of all class hierarchies
The Object Class Methods

- The `Object` class contains a few useful methods, which are inherited by all classes
- I.e., the `toString()` method is defined in the `Object` class
- Every time we define the `toString` method, we are actually **overriding** an inherited definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the objects class along with some other information
- Also in `Object`:
  - `equals()` returns `T` if and only if __________
  - `clone()` returns __________

Abstract Classes

Forcing children classes to take responsibility and define their own methods
Need for Abstract Classes

Abstract Classes

* An abstract class is a placeholder in a class hierarchy that represents a generic concept
* An abstract class cannot be instantiated
* To declare a class as abstract:

```java
public abstract class Shape {
    // contents
}
```

* Abstract classes are an important element of software design: they allow us to establish common elements in a hierarchy that are too generic to instantiate
Abstract Classes: Rules

- An abstract class often contains abstract methods with **no definitions**
  - The abstract modifier **must** be applied to each abstract method

- An abstract class typically contains non-abstract methods with full definitions

- A class declared as abstract **does not have to** contain abstract methods –
  - simply declaring it as abstract makes the class abstract

- The child of an abstract class **must override** the abstract methods of the parent, or it, too, will be considered abstract

- An abstract method **cannot** be defined as final or static
Polymorphism

Inheritance provides Power to OOP
Polymorphism provides flexibility through inheritance

Polymorphism via Inheritance

```java
Rectangle myShape = new Rectangle();
myShape.area();
Square perfect = new Square();
myShape = perfect;
myShape.area();
```

Class `Rectangle` has a method called `area()`, and the child class `Square` overrides it.

Now consider the following invocation:
```
myShape.area();
```
Which `area()` is invoked?

If `myShape` refers to a `Rectangle` object, it invokes the `Rectangle` version of `area()`.

If `myShape` refers to a `Square` object, it invokes the `Square` version of `area()`!
Static and Dynamic Binding

- Consider the following method invocation:
  ```java
  myShape.area();
  ```

- At some point, this invocation is *bound* to the definition of the method that it invokes
  - If this binding occurred *statically* at *compile* time, then that line of code would call the same method every time

- Java defers method binding until *run* time: this is called *dynamic binding* or *late binding*

- Dynamic binding provides *flexibility* in program design

Polymorphism: "having many forms"

- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time

- Suppose we create the following reference variable
  ```java
  Rectangle myShape;
  ```

- Java allows this reference to point to a Rectangle object, or to any object of *any compatible type!*

- This *compatibility* can be established using *inheritance* or using *interfaces*
Firm.java

```java
/**
 * Demonstrates polymorphism via inheritance.
 * @author Java Foundations
 */
public class Firm {
  /**
   * Creates a staff of employees for a firm and pays them.
   */
  public static void main(String[] args) {
    Staff personnel = new Staff();
    personnel.payday();
  }
}
```

Exploring the benefits and flexibility of polymorphism

A Program that pays various types of employees using a polymorphic method
Staff.java

/**
 * Represents the personnel staff of a particular business.
 * @author Java Foundations
 */

public class Staff {
    private StaffMember[] stafflist;

    /**
     * Constructor: Sets up the list of staff members.
     */
    public Staff () {
        stafflist = new StaffMember[3];
        stafflist[0] = new Executive("Tony", "123 Main Line", "555-0409", "123-45-6789", 2423.07);
        stafflist[3] = new Hourly("Michael", "678 Fifth Ave.", "555-0000", "958-47-3625", 10.55);

        ((Executive)stafflist[0]).awardBonus (500.00);
        ((Hourly)stafflist[3]).addHours (40);
    }
}

Staff.java

/**
 * Pays all staff
 */

public void payday () {
    double amount;
    for (int count=0; count < stafflist.length; count++) {
        System.out.println (stafflist[count]);
        amount = stafflist[count].pay(); // polymorphic
        if (amount == 0.0)
            System.out.println ("Thanks!");
        else
            System.out.println ("Paid: " + amount);
    }
    System.out.println ("------------------------------------");
}

StaffMember.java

```java
/**
 * Represents a generic staff member
 * @author Java Foundations
 */
abstract public class StaffMember {
    protected String name;
    protected String address;
    protected String phone;

    /**
     * Constructor: Sets up this staff member using the specified information.
     */
    public StaffMember (String eName, String eAddress, String ePhone) {
        name = eName;
        address = eAddress;
        phone = ePhone;
    }

    /**
     * Derived classes must define the pay method for each type
     */
    public abstract double pay();
    --
    toString() omitted...
}
```

Volunteer.java

```java
/**
 * Represents a staff member that works as a volunteer.
 * @author Java Foundations
 */
public class Volunteer extends StaffMember {
    /**
     * Constructor: Sets up this volunteer using the specified information.
     */
    public Volunteer (String eName, String eAddress, String ePhone) {
        super (eName, eAddress, ePhone);
    }

    /**
     * @return a zero pay value for this volunteer.
     */
    public double pay() {
        return 0.0;
    }
    --
```
/**
 * Represents a general paid employee.
 * @author Java Foundations
 */

public class Employee extends StaffMember {
    protected String socialSecurityNumber;
    protected double payRate;

    /**
     * Constructor: Sets up this employee with the specified information.
     */
    public Employee (String eName, String eAddress, String ePhone,
                     String socSecNumber, double rate) {
        super (eName, eAddress, ePhone);
        socialSecurityNumber = socSecNumber;
        payRate = rate;
    }

    /**
     * @return the pay rate for this employee.
     */
    public double pay() {
        return payRate;
    }
}

/**
 * Represents an executive staff member, who can earn a bonus.
 * @author Java Foundations
 */

public class Executive extends Employee {
    private double bonus;

    /**
     * Constructor: Sets up this executive with the specified information.
     */
    public Executive (String eName, String eAddress, String ePhone,
                      String socSecNumber, double rate) {
        super (eName, eAddress, ePhone, socSecNumber, rate);
        bonus = 0;  // bonus has yet to be awarded
    }

    /**
     * Computes and returns the pay for an executive, which is the
     * regular employee payment plus a one-time bonus.
     */
    public double pay() {
        double payment = super.pay() + bonus;
        bonus = 0;
        return payment;
    }
}
/**
 * Represents an employee that gets paid by the hour.
 * Author: Java Foundations
 */

public class Hourly extends Employee {
    private int hoursWorked;

    /**
     * Constructor: Sets up this hourly employee using the specified information.
     */
    public Hourly(String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate) {
        super(eName, eAddress, ePhone, socSecNumber, rate);
        hoursWorked = 0;
    }

    /**
     * Computes and returns the pay for this hourly employee.
     */
    public double pay() {
        double payment = payRate * hoursWorked;
        hoursWorked = 0;
        return payment;
    }
}
### Exceptions and IO

**Exceptions: What to do when things go bad**

**IO: Where things often go bad**

You have been coding for a while and you may have encountered some exceptions. Here are some of them:

- Division by 0 in computing expression (ArithmeticException)
- Array index out of bounds (IndexOutOfBoundsException)
- Null pointer cannot be followed (NullPointerException)
- Generic I/O problems (e.g., no space on disk to save file, file not found, etc) (IOException)
- No permissions to save a file on the disk (FileNotFoundException)

An *exception* is an object describing unusual or erroneous situation

(An *error* is also an object, but it represents a unrecoverable situation and should not be caught)

---

**10.1 – Exceptions vs Errors**

- You have been coding for a while and you may have encountered some exceptions. Here are some of them:
  - Division by 0 in computing expression
  - Array index out of bounds
  - Null pointer cannot be followed
  - Generic I/O problems (e.g., no space on disk to save file, file not found, etc)
  - No permissions to save a file on the disk

- An *exception* is an object describing unusual or erroneous situation

- (An *error* is also an object, but it represents a unrecoverable situation and should not be caught)
10.3 – The try Statement

* Exceptions are **thrown** by a program, and may be **caught** and **handled** by another part of the program.

* To handle an exception, the line that throws the exception is executed within a **try block**.

* A try block is followed by one or more **catch clauses**.

* When an exception occurs, processing continues at the first catch clause that matches the exception type.

```java
// here is code that
// should generate no exceptions
try {
    // code to monitor
    // several possible things
    // that can go wrong
    // goes here
} catch (ExceptionTypeA ex) {
    // handler for ExceptionTypeA
} catch (ExceptionTypeB ex) {
    // handler for ExceptionTypeB
} // after a catch, continue here
```

---

**Using Exceptions in an “exceptional” way ;-)**

```java
// Counts the number of product codes that are entered
// with a zone of R and district greater than 2000.

zone = code.charAt(9);
district = Integer.parseInt(code.substring(3, 7));
valid++;
if (zone == 'R' && district > 2000) banned++;
```
The **throws** clause

```java
import java.io.*;

public class TestData
{
    // ---
    // It will read/write to a file and things can go bad!
    // ---

    public static void main(String[] args) throws IOException
    {
        String file = "test.dat";

        // More on IO shortly...
        FileWriter fw = new FileWriter(file);
        BufferedWriter bw = new BufferedWriter(fw);
        PrintWriter outFile = new PrintWriter(bw);
```
10.5 – An exception is either checked or unchecked

- A **checked exception** requires explicit handling. It **must**
  - or

- The compiler will issue error if a checked exception is **not caught** or **asserted** in a throws clause

- An **unchecked exception** does not require explicit handling (but try to catch)

- The only unchecked Java exceptions are objects of type **RuntimeException** (or any of its descendants)

- Errors are similar to RuntimeException and its descendants in the sense that
  - Errors cannot be caught
  - Errors do not require a throws clause

---

IO with Scanner and PrintWriter

Great resource! Learn and Reuse
Displaying contents read in from keyboard

/* Read in lines of text from the keyboard, 
* and print out each line after it is read in. 
* Stop when the user hits CONTROL-D. 
*/
public static void displayKeyboardInput () {
    // will not throw
    Scanner keyboardScan = new Scanner (System.in);
    do {
        String line = keyboardScan.nextLine();
        System.out.println(line);
    } while (keyboardScan.hasNext());
}

Displaying the contents of a file

/* Read in the contents of a file line by line, 
* and print out each line after it is read in. 
* Stop when the end of the file is reached. 
*/
public static void displayFile (String inFileName) {
    try {
        Scanner fileScan = new Scanner (new File(inFileName));
        while (fileScan.hasNext()) {
            String line = fileScan.nextLine();
            System.out.println(line);
        }
    } catch (IOException ex) {
        System.out.println(ex);
    }
}
Displaying the contents of a web page

/* Read in the contents of a web page line by line, *
* and print out each line after it is read in. *
* Stop when the end of the web page is reached. */

public static void displayWebPage (String urlName) {
    try {
        URL u = new URL(urlName);
        Scanner urlScan = new Scanner( u.openStream() );
        while (urlScan.hasNext()) {
            String line = urlScan.nextLine();
            System.out.println(line);
        }
    }
    catch (IOException ex) {
        System.out.println(ex);
    }
}

Writing to a File

/* Copies an input file to an output file. Displays an *
* error message if the output file cannot be created. */

public static void copyFile(String inFileName, String outFileName) {
    try {
        Scanner reader = new Scanner (new File(inFileName));
        PrintWriter writer = new PrintWriter (new File(outFileName));

        while (reader.hasNext()) {
            // Read and write line to output file
            writer.println(reader.nextLine());
        }
    }
    catch (IOException ex) {
        System.out.println(ex);  // Handle file-not-found
    }
}
Counting Characters and Lines

Write a method that takes the name of a file as input and prints out the number of characters in the file and the number of lines in the file.

```java
public static void countCharsAndLines(String filename) {
}
```
Reading File Contents Into a String

/*
 * Reads in a file and stores the contents in a String. This method is
 * inefficient because it uses a String concatenation rather than a
 * StringBuilder to collect the lines of the files
 */
public static String fileToString_inefficient (String inFileName) {
    try {
        Scanner reader = new Scanner(new File(inFileName));
        String linesFromFile = ""; // Var for accumulating String from file
        while (reader.hasNext()) { // Continue until reach end of input file
            linesFromFile = linesFromFile + reader.nextLine() + "\n";
            // nextLine() omits the newline character, so add back in
        }
        reader.close(); // Close the file reader
        return linesFromFile;
    } catch (FileNotFoundException ex) {
        System.out.println(ex); // Handle FNF by displaying message
        return ""; // Return the empty string if file not found
    }
}

Reading File Into String (More Efficient)

/* Reads in a file and stores the contents in a StringBuffer.
 * This method is more efficient because it uses a
 * StringBuilder rather than String concatenation to collect
 * the lines of the files. */
public static String fileToString (String inFileName) {
    try {
        Scanner reader = new Scanner(new File(inFileName));
        // Accumulate lines in StringBuilder
        StringBuilder builder = new StringBuilder();
        while (reader.hasNext()) { // Continue until EOF
            builder.append(reader.nextLine());
            builder.append("\n"); // nextLine() omits newline, re-add
        }
        reader.close(); // Close the file reader
        return builder.toString();
    } catch (FileNotFoundException ex) {
        System.out.println(ex); // Handle FNF by displaying message
        return ""; // Return the empty string if file not found
    }