Inheritance

The Power of OO Programming!

Geometric Shapes

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

Diagram of inheriting shapes:
- Abstract Shape
  - Circle
  - Triangle
  - Rectangle
  - Square
Inheritance

- Inheritance is a fundamental technique used to
- The child is a more specific version of the 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
Words.java

```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 definitions: " + webster.getDefinitions());
        System.out.println("Definitions per page: " + webster.computeRatio());
    }
}
```

Book.java

```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 as the parent class.

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

- It provides more **encapsulation** than visibility, but is not as tightly encapsulated as 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 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;
    }
}
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.");
    }
}
```

Advice.java

```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();
    }
}
```
Messages.java

```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:
  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.

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.
An Alternative Class Hierarchy

There is no single class hierarchy that is appropriate for all situations

The Object Class

- 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 object's class along with some other information
- Also in Object:
  - `equals()` returns T if and only if __________
  - `clone()` returns __________

Need for Abstract Classes

Abstract classes and methods are shown in italics font.
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 it so.
- 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.