Chapter 8: Inheritance

Java Software Solutions
Foundations of Program Design
Sixth Edition

by Lewis and Loftus
Inheritance

• *Inheritance* allows a software developer to derive a new class from an existing one

• The existing class is called the *parent class*, or *superclass*, or *base class*

• The derived class is called the *child class* or *subclass*

• As the name implies, the child inherits characteristics of the parent
  – methods
  – attributes (data)
Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```java
public class Vehicle {
    private int numPassengers;

    public int getNumPass() {
        return numPassengers;
    }

    public void setNumPass(int n) {
        numPassengers = n;
    }
}

public class Car extends Vehicle {
    // class contents
}

public class Main {
    ...
    Car myCar = new Car();
    myCar.setNumPass(4);
    ...
    int n = myCar.getNumPass();
}
```
Inheritance

• Methods
  – All public and protected methods from your parent’s class are available to the child class
    • Including ALL ancestors – parents, grandparents, great grandparents, etc…

• Attributes
  – All public and protected attributes from your parent’s class are available to the child class
    • Include ALL ancestors
Inheritance versus Aggregation

• Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent.

```java
public class Car extends Vehicle {
    private Wheel frontTire;
}
```

• Aggregation is a “has-a” relationship. A class Aggregation means an attribute of one class is another class. One class “has” another class.

• Car *has a* Wheel

Know the difference!
Inheritance: Why use it?

- **Software reuse** – you can reuse most of another class while just modifying a small part you care about

- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software
Visibility Revisited

- Visibility of attributes and methods of a class can be
  - **Public**: All parts of the application can use it
  - **Protected**: Can be used only in the same package and from subclasses in other packages
  - **Default (no modifier)**: Can be used only in the same package
  - **Private**: Can be used only in the same class

Whenever possible: use private!
The super Reference

- super can be used to explicitly call methods or constructors in your parent’s class
- It is always a good practice to call your parent’s constructor to setup the parent’s part of the object

```java
public class Car extends Vehicle {
    public Car() {
        super(); // call constructor in parent that takes no parameters
        .. // do other stuff
    }

    public Car(int numWheels) {
        super("A car"); // Call the constructor in the parent that take a String parameter
    }
}
```

I’m powerful enough to call my parents!
The super Reference

• A child’s constructor is responsible for calling the parent’s constructor

• The first line of a child’s constructor should use the `super` reference to call the parent’s constructor

• The `super` reference can also be used to reference other variables and methods defined in the parent’s class
Multiple Inheritance

- Java supports **single inheritance**, meaning that a derived class can have only one parent class.
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents.
- Collisions, such as the same variable name in two parents, have to be resolved.
- **Java does not support multiple inheritance**.
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead.

Coming up: Overriding Methods
Overriding Methods

- A child class can override the definition of an inherited method in favor of its own.
- The new method must have the same signature as the parent's method, but can have a different body.
- The type of the object executing the method determines which version of the method is invoked.

See Messages.java
See Thought.java
See Advice.java

Coming up: Overriding
Overriding

• A method in the parent class can be invoked explicitly using the `super` reference

• If a method is declared with the `final` modifier, it cannot be overridden

• You can also override data by declaring an attribute with the same name as a parent’s attribute: this is called *shadowing variables*

• **Shadowing variables should be avoided** because it tends to cause unnecessarily confusing code

Coming up: Overloading vs. Overriding
Overloading vs. Overriding

- **Overloading:**
  - multiple methods in one class with same name, but different signatures
  - defines similar operation on different parameter sets

- **Overriding:**
  - method in a child class with same signature as method in parent's class.
  - Defines a similar operation in different ways for different object types (parent versus child)
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

- The **Object** class contains a few useful methods, which are inherited by all classes.

- For example, 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.
The Object Class

• The `equals` method of the `Object` class returns true if two references are aliases

• We can override `equals` in any class to define equality in some more appropriate way

• As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters

• The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version

Coming up: 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.

• We use the modifier abstract on the class header to declare a class as abstract:

```java
public abstract class Product {
    // contents
}
```
Abstract Classes

• An abstract class often contains abstract methods with no definitions (like an interface)

• Unlike an interface, the abstract modifier must be applied to each abstract method

• Also, 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
Abstract Classes

• 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

• The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too generic to instantiate
Abstract Class Example

• Lets create an abstract class
  – Shape
    – Has a type attribute and associated setters/getters
    – Defines a getArea method
  – Circle extends Shape
  – Rectangle extends Shape
Inheritance Design Issues

• Every derivation should be an is-a relationship

• Think about the potential future of a class hierarchy, and design classes to be reusable and flexible

• Find common characteristics of classes and push them as high in the class hierarchy as appropriate

• Override methods as appropriate to tailor or change the functionality of a child

• Add new attributes to children, but don't redefine (shadow) inherited attributes
Inheritance Design Issues

• Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data

• Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions

• Use abstract classes to represent general concepts that lower classes have in common

• Use visibility modifiers carefully to provide needed access without violating encapsulation

Coming up: Restricting Inheritance
Restricting Inheritance

- The **final** modifier can be used to curtail inheritance.
- If the **final** modifier is applied to a method, then that method cannot be overridden in any descendent classes.
- If the **final** modifier is applied to an entire class, then that class cannot be used to derive any children at all.