Chapter 7: Inheritance

Presentation slides for

Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley

Inheritance

- **Another fundamental object-oriented technique is called inheritance, which enhances software design and promotes reuse**

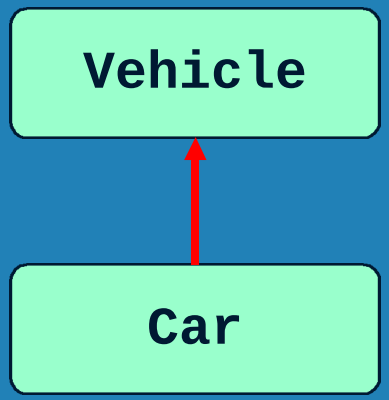
- **Chapter 7 focuses on:**
  - **deriving new classes**
  - **creating class hierarchies**
  - **the `protected` modifier**
  - **polymorphism via inheritance**
  - **inheritance used in graphical user interfaces**

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**
- **That is, the child class inherits the methods and data defined for the parent class**

Inheritance

- **Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class**



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

Deriving Subclasses

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

```
class Car extends Vehicle
{
   // class contents
}
```

- See Words.java (page 324)
- See Book.java (page 325)
- See Dictionary.java (page 326)

Controlling Inheritance

- Visibility modifiers determine which class members get inherited and which do not
- Variables and methods declared with `public` visibility are inherited, and those with `private` visibility are not
- But `public` variables violate our goal of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The `protected` Modifier

- The `protected` visibility modifier allows a member of a base class to be inherited into the child
- But `protected` visibility provides more encapsulation than `public` does
- However, `protected` visibility is not as tightly encapsulated as `private` visibility
- The details of each modifier are given in Appendix F

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 `super` reference can be used to refer to the parent class, and is often used to invoke the parent's constructor
- See Words2.java (page 328)
- See Book2.java (page 329)
- See Dictionary2.java (page 330)

Single vs. 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**
- **In most cases, the use of interfaces gives us the best aspects of multiple inheritance without the overhead**

Overriding Methods

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

Overriding Methods

- See Messages.java (page 332)
- See Thought.java (page 333)
- See Advice.java (page 334)

- Note that a parent method can be explicitly invoked using the `super` reference

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

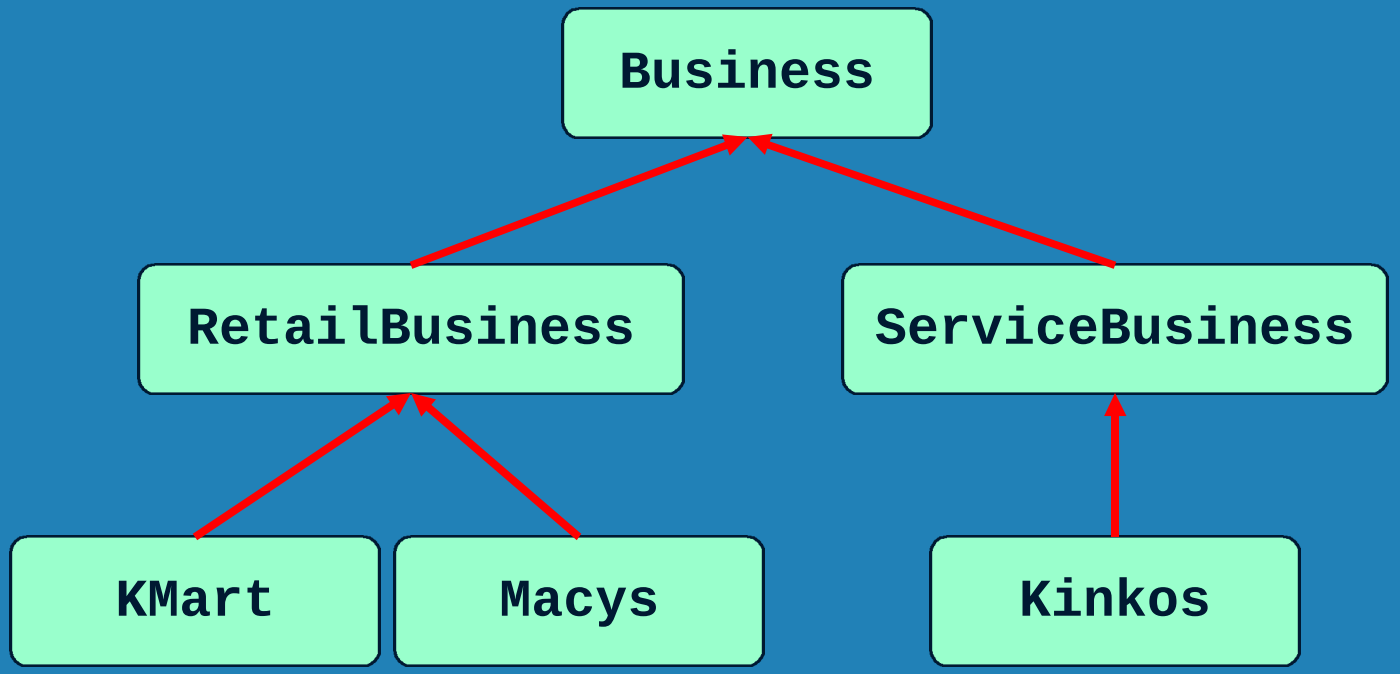
- The concept of overriding can be applied to data (called *shadowing variables*), there is generally no need for it

Overloading vs. Overriding

- **Don't confuse the concepts of overloading and overriding**
- **Overloading deals with multiple methods in the same class with the same name but 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 data**
- **Overriding lets you define a similar operation in different ways for different object types**

Class Hierarchies

- **A child class of one parent can be the parent of another child, forming *class hierarchies***



Class Hierarchies

- **Two children of the same parent are called *siblings***
- **Good class design puts all common features as high in the hierarchy as is reasonable**
- **An inherited member is continually passed down the line**
- **Class hierarchies often have to be extended and modified to keep up with changing needs**
- **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
- The `Object` class is therefore 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 have defined `toString`, we have actually been overriding it
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class and a hash value

The `Object` Class

- That's why the `println` method can call `toString` for any object that is passed to it – all objects are guaranteed to have a `toString` method via inheritance

- See Academia.java (page 339)
- See Student.java (page 340)
- See GradStudent.java (page 341)

- The equals method of the Object class determines if two references are aliases

- You may choose to override `equals` to define equality in some other way

Abstract Classes

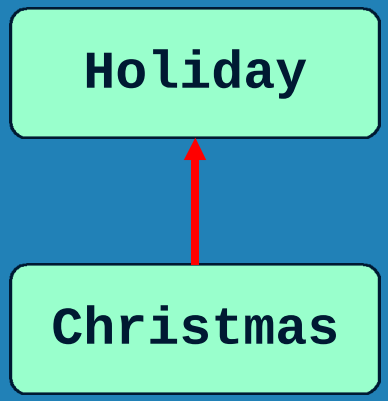
- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract
- An abstract class often contains abstract methods (like an interface does), though it doesn't have to

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 (because it must be overridden) or static (because it has no definition yet)
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is to general to instantiate

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Holiday` class is used to derive a child class called `Christmas`, then a `Holiday` reference could actually be used to point to a `Christmas` object




```
Holiday day;
day = new Christmas();
```

References and Inheritance

- **Assigning a predecessor object to an ancestor reference is considered to be a widening conversion, and can be performed by simple assignment**
- **Assigning an ancestor object to a predecessor reference can also be done, but it is considered to be a narrowing conversion and must be done with a cast**
- **The widening conversion is the most useful**

Polymorphism via Inheritance

- We saw in Chapter 5 how an interface can be used to create a *polymorphic reference*
- Recall that a polymorphic reference is one which can refer to different types of objects at different times
- Inheritance can also be used as a basis of polymorphism
- An object reference can refer to one object at one time, then it can be changed to refer to another object (related by inheritance) at another time

Polymorphism via Inheritance

- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrode it
- Now consider the following invocation:

```
day.celebrate();
```

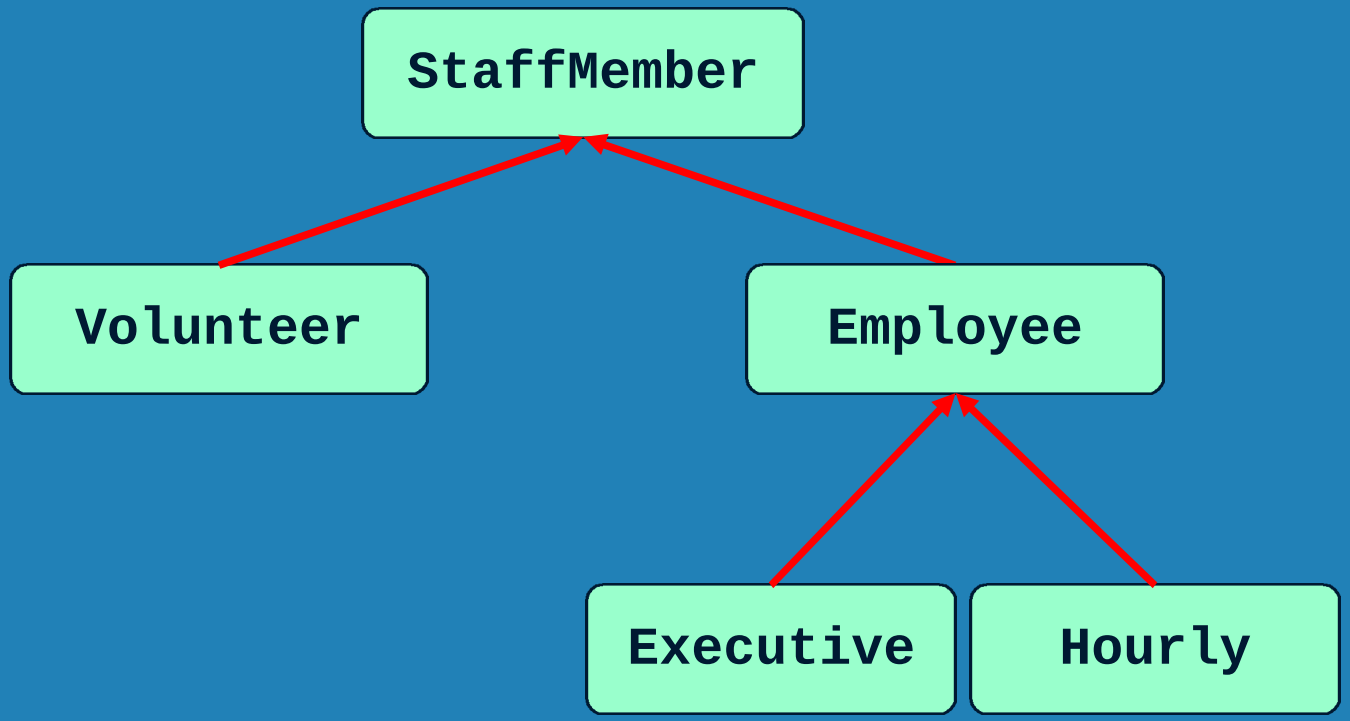
- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

Polymorphism via Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Note that, if an invocation is in a loop, the exact same line of code could execute different methods at different times
- Polymorphic references are therefore resolved at run-time, not during compilation

Polymorphism via Inheritance

- **Consider the following class hierarchy:**



Polymorphism via Inheritance

- **Now consider the task of paying all employees**

- **See Firm.java (page 345)**
- **See Staff.java (page 346)**
- **See StaffMember.java (page 348)**
- **See Volunteer.java (page 349)**
- **See Employee.java (page 351)**
- **See Executive.java (page 352)**
- **See Hourly.java (page 353)**

Indirect Access

- An inherited member can be referenced directly by name in the child class, as if it were declared in the child class
- But even if a method or variable is not inherited by a child, it can still be accessed indirectly through parent methods
- See FoodAnalysis.java (page 355)
- See FoodItem.java (page 356)
- See Pizza.java (page 357)

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- One interface can be used as the parent of another
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the parent and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (the do not overlap)

Applets and Inheritance

- An applet is an excellent example of inheritance
- Recall that when we define an applet, we extend the `Applet` class
- The `Applet` class already handles all the details about applet creation and execution, including the interaction with a web browser
- Our applet classes only have to deal with issues that specifically relate to what our particular applet will do

Extending Event Adapter Classes

- In Chapter 5 we discussed the creation of listener classes by implementing a particular interface (such as `MouseListener` interface)
- A listener can also be created by extending a special *adapter class* of the Java class library
- Each listener interface has a corresponding adapter class (such as the `MouseAdapter` class)
- Each adapter class implements the corresponding listener and provides empty method definitions

Extending Event Adapter Classes

- When you derive a listener class from an adapter class, you override any event methods of interest (such as the `mouseClicked` method)
- Note that this avoids the need to create empty definitions for unused events
- See OffCenter.java (page 360)

GUI Components

- A *GUI component* is an object that represents a visual entity in an graphical user interface (such as a button or slider)
- Components can generate events to which listener objects can respond
- For example, an applet is a component that can generate mouse events
- An applet is also a special kind of component, called a *container*, in which other components can be placed

GUI Components

- See Fahrenheit.java (page 363)
- Components are organized into an inheritance class hierarchy so that they can easily share characteristics
- When we define certain methods, such as the `paint` method of an applet, we are actually overriding a method defined in the `Component` class, which is ultimately inherited into the `Applet` class
- See Doodle.java (page 367)
- See DoodleCanvas.java (page 369)