Chapter 7: Inheritance

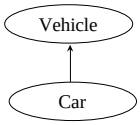
- Another fundamental object-oriented technique is called inheritance, which, when used correctly, supports reuse and enhances software designs
- Chapter 7 focuses on:
  - the concept of inheritance
  - inheritance in Java
  - the `protected` modifier
  - adding and modifying methods through inheritance
  - creating class hierarchies

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
- In programming, the child class inherits the methods and data defined for the parent class

Inheritance

- Inheritance relationships are often shown graphically, 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, the reserved word `extends` is used to establish an inheritance relationship

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

- See `Words.java`

The `protected` Modifier

- The 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 goal of encapsulation
- The `protected` visibility modifier allows a member to be inherited, but provides more protection than `public` does
- 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`

Overriding Methods

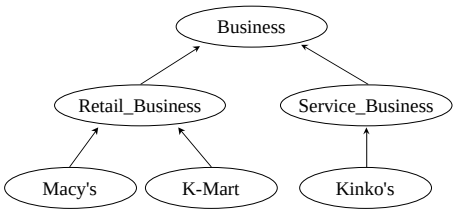
- A child class can *override* the definition of an inherited method in favor of its own
- That is, a child can redefine a method it inherits from its parent
- The new method must have the same signature as the parent's method, but can have different code in the body
- The object type determines which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

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,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
- 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

- All objects 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 contains a few useful methods, such as `toString()`, which are inherited by all classes
- See `Academia.java`

Abstract Classes

- The modifier `abstract` is used to define abstract classes and methods
- The children of the abstract class are expected to define implementations for the abstract methods in ways appropriate for them
- If a child class does not define all abstract methods of the parent, then the child is also abstract
- An abstract class is often too generic to be of use by itself

Abstract Classes

- An *abstract class* cannot be instantiated
- It is used in a class hierarchy to organize common features at appropriate levels
- An *abstract method* has no implementation, just a name and signature
- An abstract class often contains abstract methods
- Any class that contains an abstract method is by definition abstract

Abstract Classes

- An abstract method cannot be declared as `final`, because it must be overridden in a child class
- An abstract method cannot be declared as `static`, because it cannot be invoked without an implementation
- Abstract classes are placeholders that help organize information and provide a base for polymorphic references

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

- A *polymorphic reference* is one which can refer to one of several possible methods
- 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 `Holiday`'s version of `celebrate`; if it refers to a `Christmas` object, it invokes that version

Polymorphism

- In general, 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

- Note that, because all classes inherit from the `Object` class, an `Object` reference can refer to any type of object
- A `Vector` is designed to store `Object` references
- The `instanceOf` operator can be used to determine the class from which an object was created

The `super` Reference Revisited

- The `super` reference can be used to invoke any method from the parent class
- This ability is often helpful when using overridden methods
- The syntax is:

```
super.method(parameters)
```

- See `Firm.java`

Defined vs. Inherited

- A subtle feature of inheritance is the fact that even if a method or variable is not inherited by a child, it is still *defined* for that child
- An inherited member can be referenced directly in the child class, as if it were declared in the child class
- But even members that are not inherited exist for the child, and can be referenced indirectly through parent methods
- See `FoodAnalysis.java`

Interfaces

- We've used the term interface to mean the set of service methods provided by an object
- That is, the set of methods that can be invoked through an object define the way the rest of the system interacts, or interfaces, with that object
- The Java language has an interface construct that formalizes this concept
- A Java *interface* is a collection of constants and abstract methods

Interfaces

- A class that *implements* an interface must provide implementations for all of the methods defined in the interface
- This relationship is specified in the header of the class:

```
class class-name implements interface-name {
}
```


Interfaces

- An interface can be implemented by multiple classes
- Each implementing class can provide their own unique version of the method definitions
- An interface is not a class, and cannot be used to instantiate an object
- An interface is not part of the class hierarchy
- A class can be derived from a base class and implement one or more interfaces

Interfaces

- Unlike interface methods, interface constants require nothing special of the implementing class
- Constants in an interface can be used in the implementing class as if they were declared locally
- This feature provides a convenient technique for distributing common constant values among multiple classes

Interfaces

- An interface can be derived from another interface, using the `extends` reserved word
- The child interface inherits the constants and abstract methods of the parent
- Note that the interface hierarchy and the class hierarchy are distinct
- A class that implements the child interface must define all methods in both the parent and child

Interfaces

- An interface name can be used as a generic reference type name
- A reference to any object of any class that implements that interface is compatible with that type
- For example, if `Philosopher` is the name of an interface, it can be used as the type of a parameter to a method
- An object of any class that implements `Philosopher` can be passed to that method

Interfaces

- Note the similarities between interfaces and abstract classes
- Both define abstract methods that are given definitions by a particular class
- Both can be used as generic type names for references
- However, a class can implement multiple interfaces, but can only be derived from one class

Interfaces

- A class that implements multiple interfaces specifies all of them in its header, separated by commas
- The ability to implement multiple interfaces provides many of the features of *multiple inheritance*, the ability to derive one class from two or more parents
- Java does not support multiple inheritance
- An applet is a good example of inheritance
- See `OffCenter.java`

GUI Components

- There are several GUI components that permit specific kinds of user interaction:
  - labels
  - text fields
  - text areas
  - lists
  - buttons
  - scrollbars
  - Canvas

Labels

- A *label* defines a line of text displayed on a GUI
- Labels are static in the sense that they cannot be selected or modified by the human user once added to a container
- A label is instantiated from the `Label` class
- The `Label` class contains several constructors and methods for setting up and modifying a label's content and alignment

Text Fields and Text Areas

- A *text field* displays a single line of text in a GUI
- It can be made editable, and provide a means to get input from the user
- *text area* is similar, but displays multiple lines of text
- They are defined by the `TextField` and `TextArea` classes
- A text area automatically has scrollbars on its bottom and right sides
- See `Fahrenheit.java`

Lists

- A *list,* in the Java GUI sense, is used to display a list selectable strings
- A list component can contain any number of strings and can be instantiated to allow multiple selections within it
- The size of the list is specified by the number of visible rows or strings within it
- scrollbar will automatically appear on the right side of a list if the number of items exceed the visible area
- A list is defined by the `List` class

Buttons

- The `java.awt` package supports four distinct types of buttons:
  - Push buttons
  - Choice Buttons
  - Checkbox buttons
  - Radio buttons
- Each button type serves a particular purpose

Push Button

- A *push button* is a single button which can be created with or without a label
- A system is usually designed such that when a push button is pressed, a particular action occurs
- It is defined by the `Button` class
- See `Doodle.java`

Choice button

- A *choice button* is a single button which displays a list of choices when pushed
- The user can then scroll through and choose the appropriate option
- The current choice is displayed next to the choice button
- It is defined by the `Choice` class

Checkbox button

- A *checkbox button* can be toggled on or off
- A set of checkbox buttons are often used to define a set of options as a group, though one can be used by itself
- If used in a group, more than one option can be chosen at any one time
- Defined by the `Checkbox` class

Radio buttons

- A *radio button*, like a checkbox button, is toggled on or off
- Radio buttons must be grouped into a set, and only one button can be selected at any one time
- When one button of a group is selected, the currently selected button in that group is automatically reset
- They are used to select among a set of mutually exclusive options
- Radio button sets are defined by the `Checkbox` and `CheckboxGroup` classes

Scrollbars

- A *scrollbar* is a slider that indicates a relative position or quantity
- They are automatic on text areas and list components, but can be used independently
- The position of the slider in the range corresponds to a particular numeric value in a range associated with the scrollbar
- A scrollbar is defined by the `Scrollbar` class