Another fundamental object-oriented technique is inheritance, used to organize and create reusable classes.

Chapter 7 focuses on:

- deriving new classes from existing classes
- creating class hierarchies
- abstract classes
- 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

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones.

- *Software reuse* is at the heart of inheritance.

- 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.
Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead 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.

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

- See `Words.java` (page 393)
- See `Book.java` (page 394)
- See `Dictionary.java` (page 395)
Visibility Modifiers

- Visibility modifiers determine which class members can be used by derived classes and which cannot.

- Variables and methods declared with `public` visibility can be used.

- Variables and methods declared with `private` visibility cannot.
Constructors cannot be used in child classes, 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 often is used to invoke the parent's constructor.

See Words2.java (page 397)

See Book2.java (page 398)

See Dictionary2.java (page 399)
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.
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 (page 401)
See Thought.java (page 402)
See Advice.java (page 403)
Overloading vs. Overriding

- Don't confuse the concepts of overloading and 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 data
- Overriding lets you define a similar operation in different ways for different object types
A child class of one parent can be the parent of another child, forming a class hierarchy.
Class Hierarchies

- 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.
- 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

- 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 an existing definition.

- The `toString` method in the `Object` class is defined to return a string that contains the name of the object’s class together along with some other information.
The Object Class

- All objects are guaranteed to have a `toString` method via inheritance
- Thus the `println` method can call `toString` for any object that is passed to it

- See `Academia.java` (page 406)
- See `Student.java` (page 407)
- See `StudentAthlete.java` (page 408)
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.

- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters.

- Therefore, the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version.
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 Whatever {
    // contents
}
```
Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface does).
- Unlike an interface, the `abstract` modifier must be applied to each abstract method.
- An abstract class typically contains non-abstract methods (with bodies), further distinguishing abstract classes from interfaces.
- A class declared as abstract does not need to contain abstract methods.
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 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 too general to instantiate.

- See Pets.java (page 411)
- See Pet.java (page 412)
- See Dog.java (page 413)
- See Snake.java (page 414)
Indirect Use of Members

- 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 directly accessible by a child, it can still be accessed indirectly through parent methods.

- See `FoodAnalysis.java` (page 416)
- See `FoodItem.java` (page 417)
- See `Pizza.java` (page 418)
Designing for Inheritance

- Inheritance should be carefully considered during software design
- Every derivation should be an is-a relationship
- Design a class hierarchy so that it can be reused in the future
- Use interfaces to create a class that serves multiple roles (simulating multiple inheritance)
- Override general methods such as `toString` and `equals` appropriately

- See page 419 for more items to keep in mind during design
Polymorphism

- A reference can be *polymorphic*, which can be defined as "having many forms"

  ```java
  obj.doIt();
  ```

- This line of code might execute different methods at different times if the object that `obj` points to changes

- Polymorphic references are resolved at run time; this is called *dynamic binding*

- Careful use of polymorphic references can lead to elegant, robust software designs

- Polymorphism can be accomplished using inheritance or using interfaces
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 be used to point to a Christmas object.

```java
Holiday day;
day = new Christmas();
```
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 be done also, but it is considered to be a narrowing conversion and must be done with a cast.

The widening conversion is the most useful.

An Object reference can be used to refer to any object.
Polymorphism via Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked.

- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrides it.

- Now consider the following invocation:

  ```java
  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.
Consider the following class hierarchy:
Now consider the task of paying all employees

See `Firm.java` (page 423)
See `Staff.java` (page 424)
See `StaffMember.java` (page 426)
See `Volunteer.java` (page 427)
See `Employee.java` (page 428)
See `Executive.java` (page 430)
See `Hourly.java` (page 431)
Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable
  ```
  Doable obj;
  ```

- The `obj` reference can be used to point to any object of any class that implements the `Doable` interface

- The version of `doThis` that the following line invokes depends on the type of object that `obj` is referencing
  ```
  obj.doThis();
  ```
Designing for Polymorphism

- During the design phase, opportunities for polymorphic solutions should be identified.

- Use polymorphism when different types of objects perform the same type of behavior.

- Identifying polymorphic opportunities comes easier with experience.
Inheritance and GUIs

- An applet is an excellent example of inheritance.

- Recall that when we define an applet, we extend the `Applet` class or the `JApplet` class.

- The `Applet` and `JApplet` classes already handle all the details about applet creation and execution, including:
  - interaction with a Web browser
  - accepting applet parameters through HTML
  - enforcing security restrictions
Inheritance and GUIs

- Our applet classes only have to deal with issues that specifically relate to what our particular applet will do.

- When we define the `paint` method of an applet, for instance, we are actually overriding a method defined in the `Component` class, which is ultimately inherited into the `Applet` or `JApplet` class.
The Component Class Hierarchy

- The Java classes that define GUI components are part of a class hierarchy.
- Swing GUI components typically are derived from the JComponent class which is derived from the Container class which is derived from the Component class.
- Many Swing components can serve as (limited) containers, because they are derived from the Container class.
Mouse Events

- Events related to the mouse are separated into *mouse events* and *mouse motion events*

- Mouse Events:
  - *mouse pressed* – the mouse button is pressed down
  - *mouse released* – the mouse button is released
  - *mouse clicked* – the mouse button is pressed down and released without moving the mouse in between
  - *mouse entered* – the mouse pointer is moved onto (over) a component
  - *mouse exited* – the mouse pointer is moved off of a component
Mouse Events

Mouse Motion Events:

- *mouse moved* – the mouse is moved
- *mouse dragged* – the mouse is dragged

To satisfy the implementation of a listener interface, empty methods must be provided for unused events.

An `ArrayList` object is used to store objects so they can be redrawn as necessary.

See `Dots.java` (page 440)

See `DotsPanel.java` (page 441)
The Dots Program
Mouse Events

- Each time the `repaint` method is called on an applet, the window is cleared prior to calling `paint`.

- *Rubberbanding* is the visual effect caused by "stretching" a shape as it is drawn using the mouse.

- See `RubberLines.java` (page 444).

- See `RubberLinesPanel.java` (page 445).
The RubberLines Program
Event Adapter Classes

- Listener classes can be created by implementing a particular interface (such as `MouseListener` interface).

- A listener also can be created by extending an event adapter class.

- 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.
Event Adapter Classes

- When we derive a listener class from an adapter class, we override any event methods of interest (such as the `mouseClicked` method).

- Empty definitions for unused event methods need not be provided.

- See `OffCenter.java` (page 448)

- See `OffCenterPanel.java` (page 449)
The OffCenter Program

Distance: 138.96

Applet started.
Chapter 7 has focused on:

• deriving new classes from existing classes
• creating class hierarchies
• abstract classes
• polymorphism via inheritance
• inheritance used in graphical user interfaces