7 Inheritance

This chapter explains inheritance, a fundamental technique for organizing and creating classes. It is a simple but powerful idea that influences the way we design object-oriented software. Furthermore, inheritance enhances our ability to reuse classes in other situations and programs. We explore how classes can be related to form inheritance hierarchies and how these relationships allow us to create polymorphic references. This chapter also revisits the concept of a formal Java interface. Finally, we discuss how inheritance affects various issues related to graphical user interfaces (GUIs) in Java.

chapter objectives

- Derive new classes from existing ones.
- Explain how inheritance supports software reuse.
- Add and modify methods in child classes.
- Discuss how to design class hierarchies.
- Define polymorphism and determine how it can be accomplished.
- Examine inheritance hierarchies for interfaces.
- Discuss the use of inheritance in Java GUI frameworks.
- Examine and use the GUI component class hierarchy.

7.0 creating subclasses

In Chapter 4 we presented the analogy that a class is to an object as a blueprint is to a house. A class establishes the characteristics and behaviors of an object but reserves no memory space for variables (unless those variables are declared as `static`). Classes are the plan, and objects are the embodiment of that plan.

Many houses can be created from the same blueprint. They are essentially the same house in different locations with different people living in them. But suppose you want a house that is similar to another but with some different or additional features. You want to start with the same basic blueprint but modify it to suit your needs and desires. Many housing developments are created this way. The houses in the development have the same core layout, but they have unique features. For instance, they might all be split-level homes with the same bedroom, kitchen, and living-room configuration, but some have a fireplace or full basement while others do not, or an attached garage instead of a carport.

It's likely that the housing developer commissioned a master architect to create a single blueprint to establish the basic design of all houses in the development, then a series of new blueprints that include variations designed to appeal to different buyers. The act of creating the series of blueprints was simplified since they all begin with the same underlying structure, while the variations give them unique characteristics that may be important to the prospective owners.

Creating a new blueprint that is based on an existing blueprint is analogous to the object-oriented concept of inheritance, which is a powerful software development technique and a defining characteristic of object-oriented programming.

derived classes

key concept

Inheritance is the process of deriving a new class from an existing one.

Inheritance is the process in which a new class is derived from an existing one. The new class automatically contains some or all of the variables and methods in the original class. Then, to tailor the class as needed, the programmer can add new variables and methods to the derived class or modify the inherited ones.

In general, new classes can be created via inheritance faster, easier, and cheaper than by writing them from scratch. At the heart of inheritance is the idea of *software reuse*. By using existing software components to create new ones, we capitalize on the effort that went into the design, implementation, and testing of the existing software.

Keep in mind that the word *class* comes from the idea of classifying groups of objects with similar characteristics. Classification schemes often use levels of classes that relate to each other. For example, all mammals share certain characteristics: They are warmblooded, have hair, and bear live offspring. Now consider a subset of mammals, such as horses. All horses are mammals and have all of the characteristics of mammals, but they also have unique features that make them different from other mammals.

key concept
One purpose of inheritance is to reuse existing software.

If we translate this idea into software terms, an existing class called `Mammal` would have certain variables and methods that describe the state and behavior of mammals. A `Horse` class could be derived from the existing `Mammal` class, automatically inheriting the variables and methods contained in `Mammal`. The `Horse` class can refer to the inherited variables and methods as if they had been declared locally in that class. New variables and methods can then be added to the derived class to distinguish a horse from other mammals. Inheritance thus nicely models many situations found in the natural world.

key concept
Inherited variables and methods can be used in the derived class as if they had been declared locally.

The original class that is used to derive a new one is called the *parent class*, *superclass*, or *base class*. The derived class is called a *child class*, or *subclass*. Java uses the reserved word `extends` to indicate that a new class is being derived from an existing class.

The derivation process should establish a specific kind of relationship between two classes: an *is-a relationship*. This type of relationship means that the derived class should be a more specific version of the original. For example, a horse is a mammal. Not all mammals are horses, but all horses are mammals.

key concept
Inheritance creates an is-a relationship between all parent and child classes.

Let's look at an example. The program shown in Listing 7.1 instantiates an object of class `Dictionary`, which is derived from a class called `Book`. In the `main` method, two methods are invoked through the `Dictionary` object: one that was declared locally in the `Dictionary` class and one that was inherited from the `Book` class.

The `Book` class (see Listing 7.2) is used to derive the `Dictionary` class (see Listing 7.3) using the reserved word `extends` in the header of `Dictionary`. The `Dictionary` class automatically inherits the definition of the `pageMessage` method and the `pages` variable. It is as if the `pageMessage` method and the `pages` variable were declared inside the `Dictionary` class. Note that the `definitionMessage` method refers explicitly to the `pages` variable.

listing 7.1


```
//********************************************************************
//  Words.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an inherited method.
//********************************************************************

public class Words
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Dictionary webster = new Dictionary ();

      webster.pageMessage();
      webster.definitionMessage();
   }
}
```

output

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35
```

Also, note that although the `Book` class is needed to create the definition of `Dictionary`, no `Book` object is ever instantiated in the program. An instance of a child class does not rely on an instance of the parent class.

Inheritance is a one-way street. The `Book` class cannot use variables or methods that are declared explicitly in the `Dictionary` class. For instance, if we created an object from the `Book` class, it could not be used to invoke the `definitionMessage` method. This restriction makes sense because a child class is a more specific version of the parent class. A dictionary has pages because all books have pages; but although a dictionary has definitions, not all books do.

Inheritance relationships are often represented in UML class diagrams. Figure 7.1 shows the inheritance relationship between the `Book` and `Dictionary` classes. An arrow with an open arrowhead is used to show inheritance in a UML diagram, with the arrow pointing from the child class to the parent class.

listing 7.2


```
//********************************************************************
//  Book.java       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance.
//********************************************************************

public class Book
{
   protected int pages = 1500;

   //-----------------------------------------------------------------
   //  Prints a message about the pages of this book.
   //-----------------------------------------------------------------
   public void pageMessage ()
   {
      System.out.println ("Number of pages: " + pages);
   }
}
```

the protected modifier

Not all variables and methods are inherited in a derivation. The visibility modifiers used to declare the members of a class determine which ones are inherited and which ones are not. Specifically, the child class inherits variables and methods that are declared public and does not inherit those that are declared private. The `pageMessage` method is inherited by `Dictionary` because it is declared with public visibility.

key concept

Visibility modifiers determine which variables and methods are inherited. Protected visibility provides the best possible encapsulation that permits inheritance.

However, if we declare a variable with public visibility so that a derived class can inherit it, we violate the principle of encapsulation. Therefore, Java provides a third visibility modifier: `protected`. Note that the variable `pages` is declared with protected visibility in the `Book` class. When a variable or method is declared with protected visibility, a derived class will inherit it, retaining some of its encapsulation properties. The encapsulation with protected visibility is not as tight as it would be if the variable or method were declared private, but it is better than if it were declared public. Specifically, a variable or method declared with protected

listing 7.3


```
//********************************************************************
//  Dictionary.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//********************************************************************

public class Dictionary extends Book
{
   private int definitions = 52500;

   //-----------------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------------
   public void definitionMessage ()
   {
      System.out.println ("Number of definitions: " + definitions);

      System.out.println ("Definitions per page: " + definitions/pages);
   }
}
```

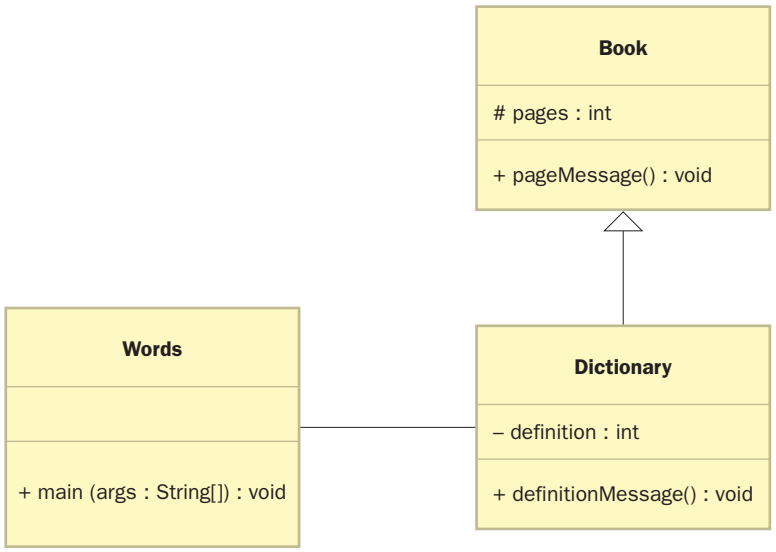


figure 7.1 A UML class diagram showing an inheritance relationship

visibility may be accessed by any class in the same package. The relationships among all Java modifiers are explained completely in Appendix F.

In a UML diagram, protected visibility can be indicated by proceeding the protected member with a hash mark (#). The `pages` variable of the `Book` class has this annotation in Fig. 7.1.

Each inherited variable or method retains the effect of its original visibility modifier. For example, the `pageMessage` method is still considered to be public in its inherited form.

Constructors are not inherited in a derived class, even though they have public visibility. This is an exception to the rule about public members being inherited. Constructors are special methods that are used to set up a particular type of object, so it wouldn't make sense for a class called `Dictionary` to have a constructor called `Book`.

the `super` reference

The reserved word `super` can be used in a class to refer to its parent class. Using the `super` reference, we can access a parent's members, even if they aren't inherited. Like the `this` reference, what the word `super` refers to depends on the class in which it is used. However, unlike the `this` reference, which refers to a particular instance of a class, `super` is a general reference to the members of the parent class.

key concept
A parent's constructor can be invoked using the `super` reference.

One use of the `super` reference is to invoke a parent's constructor. Let's look at an example. Listing 7.4 shows a modification of the original `Words` program shown in Listing 7.1. Similar to the original version, we use a class called `Book2` (see Listing 7.5) as the parent of the derived class `Dictionary2` (see Listing 7.6). However, unlike earlier versions of these classes, `Book2` and `Dictionary2` have explicit constructors used to initialize their instance variables. The output of the `Words2` program is the same as it is for the original `Words` program.

The `Dictionary2` constructor takes two integer values as parameters, representing the number of pages and definitions in the book. Because the `Book2` class already has a constructor that performs the work to set up the parts of the dictionary that were inherited, we rely on that constructor to do that work. However, since the constructor is not inherited, we cannot invoke it directly, and so we use the `super` reference to get to it in the parent class. The `Dictionary2` constructor then proceeds to initialize its `definitions` variable.

listing 7.4


```
//********************************************************************
//  Words2.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the super reference.
//********************************************************************

public class Words2
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Dictionary2 webster = new Dictionary2 (1500, 52500);

      webster.pageMessage();
      webster.definitionMessage();
   }
}
```

output

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35
```

In this case, it would have been just as easy to set the `pages` variable explicitly in the `Dictionary2` constructor instead of using `super` to call the `Book2` constructor. However, it is good practice to let each class "take care" of itself. If we choose to change the way that the `Book2` constructor sets up its `pages` variable, we would also have to remember to make that change in `Dictionary2`. By using the `super` reference, a change made in `Book2` is automatically reflected in `Dictionary2`.

A child's constructor is responsible for calling its parent's constructor. Generally, the first line of a constructor should use the `super` reference call to a constructor of the parent class. If no such call exists, Java will automatically make a call to `super()` at the beginning of the constructor. This rule ensures that a parent class initializes its variables before the child class constructor begins to execute. Using the `super` reference to invoke a parent's constructor can be done in

listing 7.5

```
//********************************************************************
//  Book2.java       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a dervied class to
//  demonstrate inheritance and the use of the super reference.
//********************************************************************

public class Book2
{
   protected int pages;

   //-----------------------------------------------------------------
   //  Sets up the book with the specified number of pages.
   //-----------------------------------------------------------------
   public Book2 (int numPages)
   {
      pages = numPages;
   }

   //-----------------------------------------------------------------
   //  Prints a message about the pages of this book.
   //-----------------------------------------------------------------
   public void pageMessage ()
   {
      System.out.println ("Number of pages: " + pages);
   }
}
```

only the child's constructor, and if included it must be the first line of the constructor.

The `super` reference can also be used to reference other variables and methods defined in the parent's class. We discuss this technique later in this chapter.

multiple inheritance

Java's approach to inheritance is called *single inheritance*. This term means that a derived class can have only one parent. Some object-oriented languages allow a child class to have multiple parents. This approach is called *multiple inheritance* and is occasionally useful for describing objects that are in between two

listing 7.6


```
//********************************************************************
//  Dictionary2.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  the use of the super reference.
//********************************************************************

public class Dictionary2 extends Book2
{
   private int definitions;

   //-----------------------------------------------------------------
   //  Sets up the dictionary with the specified number of pages
   //  (maintained by the Book parent class) and defintions.
   //-----------------------------------------------------------------
   public Dictionary2 (int numPages, int numDefinitions)
   {
      super (numPages);

      definitions = numDefinitions;
   }

   //-----------------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------------
   public void definitionMessage ()
   {
      System.out.println ("Number of definitions: " + definitions);

      System.out.println ("Definitions per page: " + definitions/pages);
   }
}
```

categories or classes. For example, suppose we had a class `Car` and a class `Truck` and we wanted to create a new class called `PickupTruck`. A pickup truck is somewhat like a car and somewhat like a truck. With single inheritance, we must decide whether it is better to derive the new class from `Car` or `Truck`. With multiple inheritance, it can be derived from both, as shown in Fig. 7.2.

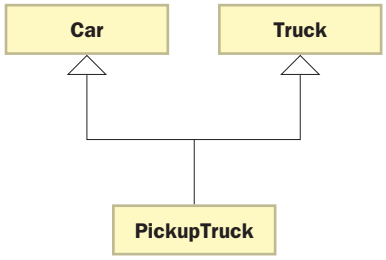


figure 7.2 A UML class diagram showing multiple inheritance

Multiple inheritance works well in some situations, but it comes with a price. What if both `Truck` and `Car` have methods with the same name? Which method would `PickupTruck` inherit? The answer to this question is complex, and it depends on the rules of the language that supports multiple inheritance.

Java does not support multiple inheritance, but interfaces provide some of the abilities of multiple inheritance. Although a Java class can be derived from only one parent class, it can implement many different interfaces. Therefore, we can interact with a particular class in particular ways while inheriting the core information from one particular parent.

7.1 overriding methods

key concept
A child class can override (redefine) the parent's definition of an inherited method.

When a child class defines a method with the same name and signature as a method in the parent class, we say that the child's version *overrides* the parent's version in favor of its own. The need for overriding occurs often in inheritance situations.

The program in Listing 7.7 provides a simple demonstration of method overriding in Java. The `Messages` class contains a `main` method that instantiates two objects: one from class `Thought` and one from class `Advice`. The `Thought` class is the parent of the `Advice` class.

Both the `Thought` class (see Listing 7.8) and the `Advice` class (see Listing 7.9) contain a definition for a method called `message`. The version of `message` defined in the `Thought` class is inherited by `Advice`, but `Advice` overrides it with an alternative version. The new version of the method prints out an entirely different message and then invokes the parent's version of the `message` method using the `super` reference.

listing 7.7


```
//********************************************************************
//  Messages.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an overridden method.
//********************************************************************

public class Messages
{
   //-----------------------------------------------------------------
   //  Instatiates two objects a 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
   }
}
```

output

```
I feel like I'm diagonally parked in a parallel universe.

Warning: Dates in calendar are closer than they appear.

I feel like I'm diagonally parked in a parallel universe.
```

The object that is used to invoke a method determines which version of the method is actually executed. When `message` is invoked using the `parked` object in the `main` method, the `Thought` version of `message` is executed. When `message` is invoked using the `dates` object, the `Advice` version of `message` is executed. This flexibility allows two objects that are related by inheritance to use the same naming conventions for methods that accomplish the same general task in different ways.

A method can be defined with the `final` modifier. A child class cannot override a final method. This technique is used to ensure that a derived class uses a particular definition for a method.

listing 7.8


```
//********************************************************************
//  Thought.java       Author: Lewis/Loftus
//
//  Represents a stray thought. Used as the parent of a derived
//  class to demonstrate the use of an overridden method.
//********************************************************************

public class Thought
{
   //-----------------------------------------------------------------
   //  Prints a message.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println ("I feel like I'm diagonally parked in a " +
                          "parallel universe.");

      System.out.println();
   }
}
```

The concept of method overriding is important to several issues related to inheritance. We explore these issues throughout this chapter.

shadowing variables

It is possible, although not recommended, for a child class to declare a variable with the same name as one that is inherited from the parent. This technique is called *shadowing variables*. It is similar to the process of overriding methods but creates confusing subtleties. Note the distinction between redeclaring a variable and simply giving an inherited variable a particular value.

Because an inherited variable is already available to the child class, there is usually no good reason to redeclare it. Someone reading code with a shadowed variable will find two different declarations that seem to apply to a variable used in the child class. This confusion causes problems and serves no purpose. A redeclaration of a particular variable name could change its type, but that is usually unnecessary. In general, shadowing variables should be avoided.

listing 7.9


```
//********************************************************************
//  Advice.java       Author: Lewis/Loftus
//
//  Represents a piece of advice. Used to demonstrate the use of an
//  overridden method.
//********************************************************************

public class Advice extends Thought
{
   //-----------------------------------------------------------------
   //  Prints a message. This method overrides the parent's version.
   //  It also invokes the parent's version explicitly using super.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println ("Warning: Dates in calendar are closer " +
                          "than they appear.");

      System.out.println();

      super.message();
   }
}
```

7.2 class hierarchies

key concept

The child of one class can be the parent of one or more other classes, creating a class hierarchy.

A child class derived from one parent can be the parent of its own child class. Furthermore, multiple classes can be derived from a single parent. Therefore, inheritance relationships often develop into *class hierarchies*. The UML class diagram in Fig. 7.3 shows a class hierarchy that incorporates the inheritance relationship between the `Mammal` and `Horse` classes.

There is no limit to the number of children a class can have or to the number of levels to which a class hierarchy can extend. Two children of the same parent are called *siblings*. Although siblings share the characteristics passed on by their common parent, they are not related by inheritance because one is not used to derive the other.

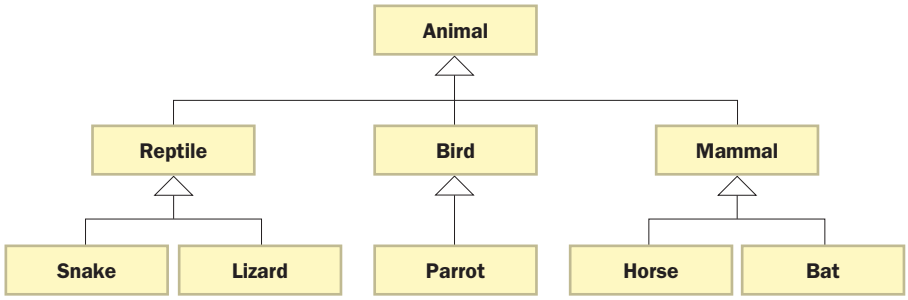


figure 7.3 A UML class diagram showing a class hierarchy

In class hierarchies, common features should be kept as high in the hierarchy as reasonably possible. That way, the only characteristics explicitly established in a child class are those that make the class distinct from its parent and from its siblings. This approach maximizes the potential to reuse classes. It also facilitates maintenance activities because when changes are made to the parent, they are automatically reflected in the descendents. Always remember to maintain the is-a relationship when building class hierarchies.

key concept
Common features should be located as high in a class hierarchy as is reasonably possible, minimizing maintenance efforts.

The inheritance mechanism is transitive. That is, a parent passes along a trait to a child class, and that child class passes it along to its children, and so on. An inherited feature might have originated in the immediate parent or possibly several levels higher in a more distant ancestor class.

There is no single best hierarchy organization for all situations. The decisions you make when you are designing a class hierarchy restrict and guide more detailed design decisions and implementation options, so you must make them carefully.

Earlier in this chapter we discussed a class hierarchy that organized animals by their major biological classifications, such as `Mammal`, `Bird`, and `Reptile`. However, in a different situation, the same animals might logically be organized in a different way. For example, as shown in Fig. 7.4, the class hierarchy might be organized around a function of the animals, such as their ability to fly. In this case, a `Parrot` class and a `Bat` class would be siblings derived from a general `FlyingAnimal` class. This class hierarchy is as valid and reasonable as the original one. The needs of the programs that use the classes will determine which is best for the particular situation.

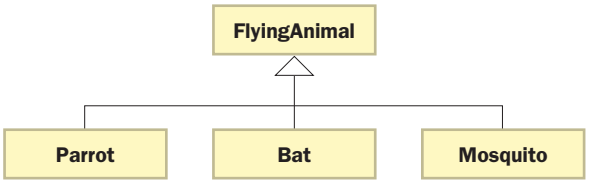


figure 7.4 An alternative hierarchy for organizing animals

the Object class

In Java, all classes are derived ultimately from the `Object` class. If a class definition doesn't use the `extends` clause to derive itself explicitly from another class, then that class is automatically derived from the `Object` class by default. Therefore, the following two class definitions are equivalent:

```
class Thing
{
   // whatever
}
```

and

```
class Thing extends Object
{
   // whatever
}
```

key concept: All Java classes are derived, directly or indirectly, from the `Object` class.

Because all classes are derived from `Object`, any public method of `Object` can be invoked through any object created in any Java program. The `Object` class is defined in the `java.lang` package of the Java standard class library. Figure 7.5 lists some of the methods of the `Object` class.

As it turns out, we've been using `Object` methods quite often in our examples. The `toString` method, for instance, is defined in the `Object` class, so the `toString` method can be called on any object. As we've seen several times, when a `println` method is called with an object parameter, `toString` is called to determine what to print.

The definition for `toString` that is provided by the `Object` class returns a string containing the object's class name followed by a numeric value that is unique for that object. Usually, we override the `Object` version of `toString` to

```
boolean equals (Object obj)
    Returns true if this object is an alias of the specified object.

String toString ()
    Returns a string representation of this object.

Object clone ()
    Creates and returns a copy of this object.
```

figure 7.5 Some methods of the `Object` class

fit our own needs. The `String` class has overridden the `toString` method so that it returns its stored string value.

The `equals` method of the `Object` class is also useful. As we've discussed previously, its purpose is to determine whether two objects are equal. The definition of the `equals` method provided by the `Object` class returns true if the two object references actually refer to the same object (that is, if they are aliases). Classes often override the inherited definition of the `equals` method in favor of a more appropriate definition. For instance, the `String` class overrides `equals` so that it returns true only if both strings contain the same characters in the same order.

key concept

The `toString` and `equals` methods are defined in the `Object` class and therefore are inherited by every class in every Java program.

Listing 7.10 shows the program called `Academia`. In this program, a `Student` object and a `GradStudent` object are instantiated. The `Student` class (see Listing 7.11) is the parent of `GradStudent` (see Listing 7.12). A graduate student is a student that has a potential source of income, such as being a graduate teaching assistant (GTA).

The `GradStudent` class inherits the variables `name` and `numCourses`, as well as the method `toString` that was defined in `Student` (overriding the version from `Object`). The `GradStudent` constructor uses the `super` reference to invoke the constructor of `Student`, and then initializes its own variables.

The `GradStudent` class augments its inherited definition with variables concerning financial support, and it overrides `toString` (yet again) to print additional information. Note that the `GradStudent` version of `toString` explicitly invokes the `Student` version of `toString` using the `super` reference.

listing 7.10


```
//********************************************************************
//  Academia.java       Author: Lewis/Loftus
//
//  Demonstrates the use of methods inherited from the Object class.
//********************************************************************

public class Academia
{
   //-----------------------------------------------------------------
   //  Creates objects of two student types, prints some information
   //  about them, then checks them for equality.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Student susan = new Student ("Susan", 5);
      GradStudent frank = new GradStudent ("Frank", 3, "GTA", 12.75);

      System.out.println (susan);
      System.out.println ();

      System.out.println (frank);
      System.out.println ();

      if (! susan.equals(frank))
         System.out.println ("These are two different students.");
   }
}
```

output

```
Student name: Susan
Number of courses: 5

Student name: Frank
Number of courses: 3
Support source: GTA
Hourly pay rate: 12.75

These are two different students.
```

listing 7.11


```
//********************************************************************
//  Student.java       Author: Lewis/Loftus
//
//  Represents a student. Used to demonstrate inheritance.
//********************************************************************

public class Student
{
   protected String name;
   protected int numCourses;

   //-----------------------------------------------------------------
   //  Sets up a student with the specified name and number of
   //  courses.
   //-----------------------------------------------------------------
   public Student (String studentName, int courses)
   {
      name = studentName;
      numCourses = courses;
   }

   //-----------------------------------------------------------------
   //  Returns information about this student as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = "Student name: " + name + "\n";

      result += "Number of courses: " + numCourses;

      return result;
   }
}
```

listing 7.12


```
//********************************************************************
//  GradStudent.java       Author: Lewis/Loftus
//
//  Represents a graduate student with financial support. Used to
//  demonstrate inheritance.
//********************************************************************

public class GradStudent extends Student
{
   private String source;
   private double rate;

   //-----------------------------------------------------------------
   //  Sets up the graduate student using the specified information.
   //-----------------------------------------------------------------
   public GradStudent (String studentName, int courses,
                       String support, double payRate)
   {
      super (studentName, courses);

      source = support;
      rate = payRate;
   }

   //-----------------------------------------------------------------
   //  Returns a description of this graduate student as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nSupport source: " + source + "\n";
      result += "Hourly pay rate: " + rate;

      return result;
   }
}
```

abstract classes

An *abstract class* represents a generic concept in a class hierarchy. An abstract class cannot be instantiated and usually contains one or more abstract methods, which have no definition. In this sense, an abstract class is similar to an interface. Unlike interfaces, however, an abstract class can contain methods that are not abstract. It can also contain data declarations other than constants.

A class is declared as abstract by including the `abstract` modifier in the class header. Any class that contains one or more abstract methods must be declared as abstract. In abstract classes (unlike interfaces) the `abstract` modifier must be applied to each abstract method. A class declared as abstract does not have to contain abstract methods.

key concept

An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.

Abstract classes serve as placeholders in a class hierarchy. As the name implies, an abstract class represents an abstract entity that is usually insufficiently defined to be useful by itself. Instead, an abstract class may contain a partial description that is inherited by all of its descendants in the class hierarchy. Its children, which are more specific, fill in the gaps.

Consider the class hierarchy shown in Fig. 7.6. The `Vehicle` class at the top of the hierarchy may be too generic for a particular application. Therefore we may choose to implement it as an abstract class. In UML diagram, abstract class names are shown in italic. Concepts that apply to all vehicles can be represented in the `Vehicle` class and are inherited by its descendants. That way, each of its descendants doesn't have to define the same concept redundantly (and perhaps inconsistently.) For example, we may say that all vehicles have a particular speed. Therefore we declare a `speed` variable in the `Vehicle` class, and all specific vehicles below it in the hierarchy automatically have that variable because of inheritance. Any change we make to the representation of the speed of a vehicle is automatically reflected in all descendant classes. Similarly, we may declare an abstract

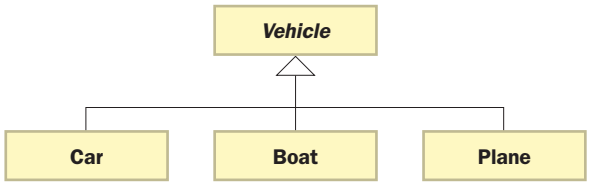


figure 7.6 A `vehicle` class hierarchy

method called `fuelConsumption`, whose purpose is to calculate how quickly fuel is being consumed by a particular vehicle. The details of the `fuelConsumption` method must be defined by each type of vehicle, but the `Vehicle` class establishes that all vehicles consume fuel and provides a consistent way to compute that value.

Some concepts don't apply to all vehicles, so we wouldn't represent those concepts at the `Vehicle` level. For instance, we wouldn't include a variable called `numberOfWheels` in the `Vehicle` class, because not all vehicles have wheels. The child classes for which wheels are appropriate can add that concept at the appropriate level in the hierarchy.

There are no restrictions as to where in a class hierarchy an abstract class can be defined. Usually they are located at the upper levels of a class hierarchy. However, it is possible to derive an abstract class from a nonabstract parent.

key concept

A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.

Usually, a child of an abstract class will provide a specific definition for an abstract method inherited from its parent. Note that this is just a specific case of overriding a method, giving a different definition than the one the parent provides. If a child of an abstract class does not give a definition for every abstract method that it inherits from its parent, the child class is also considered abstract.

Note that it would be a contradiction for an abstract method to be modified as `final` or `static`. Because a final method cannot be overridden in subclasses, an abstract final method would have no way of being given a definition in subclasses. A static method can be invoked using the class name without declaring an object of the class. Because abstract methods have no implementation, an abstract static method would make no sense.

Choosing which classes and methods to make abstract is an important part of the design process. You should make such choices only after careful consideration. By using abstract classes wisely, you can create flexible, extensible software designs. We present an example later in this chapter that relies on an abstract class to organize a class hierarchy.

7.3 indirect use of class members

There is a subtle feature of inheritance that is worth noting at this point. The visibility modifiers determine whether a variable or method is inherited into a subclass. If a variable or method is inherited, it can be referenced directly in the subclass by name, as if it were declared locally in the subclass. However, all vari-

ables and methods that are defined in a parent class exist for an object of a derived class, even though they can't be referenced directly. They can, however, be referenced indirectly.

Let's look at an example that demonstrates this situation. The program shown in Listing 7.13 contains a `main` method that instantiates a `Pizza` object and invokes a method to determine how many calories the pizza has per serving due to its fat content.

The `FoodItem` class shown in Listing 7.14 represents a generic type of food. The constructor of `FoodItem` accepts the number of grams of fat and the number of servings of that food. The `calories` method returns the number of calories due to fat, which the `caloriesPerServing` method invokes to help compute the number of fat calories per serving.

listing 7.13


```
//********************************************************************
//  FoodAnalysis.java       Author: Lewis/Loftus
//
//  Demonstrates indirect referencing through inheritance.
//********************************************************************

public class FoodAnalysis
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Pizza special = new Pizza (275);

      System.out.println ("Calories per serving: " +
                          special.caloriesPerServing());
   }
}
```

output

```
Calories per serving: 309
```

listing 7.14


```
//********************************************************************
//  FoodItem.java       Author: Lewis/Loftus
//
//  Represents an item of food. Used as the parent of a derived class
//  to demonstrate indirect referencing through inheritance.
//********************************************************************

public class FoodItem
{
   final private int CALORIES_PER_GRAM = 9;
   private int fatGrams;
   protected int servings;

   //-----------------------------------------------------------------
   //  Sets up this food item with the specified number of fat grams
   //  and number of servings.
   //-----------------------------------------------------------------
   public FoodItem (int numFatGrams, int numServings)
   {
      fatGrams = numFatGrams;
      servings = numServings;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the number of calories in this food item
   //  due to fat.
   //-----------------------------------------------------------------
   private int calories()
   {
      return fatGrams * CALORIES_PER_GRAM;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the number of fat calories per serving.
   //-----------------------------------------------------------------
   public int caloriesPerServing()
   {
      return (calories() / servings);
   }
}
```

The `Pizza` class, shown in Listing 7.15, is derived from `FoodItem class`, but it adds no special functionality or data. Its constructor calls the constructor of `FoodItem`, using the `super` reference assuming that there are eight servings per pizza.

Note that the `Pizza` object called `special` in the `main` method is used to invoke the method `caloriesPerServing`, which is defined as a public method of `FoodItem` and is therefore inherited by `Pizza`. However, `caloriesPerServing` calls `calories`, which is declared `private`, and is therefore not inherited by `Pizza`. Furthermore, `calories` references the variable `fatGrams` and the constant `CALORIES_PER_GRAM`, which are also declared with private visibility.

key concept

All members of a superclass exist for a subclass, but they are not necessarily inherited. Only inherited members can be referenced by name in the subclass.

Even though `Pizza` did not inherit `calories`, `fatGrams`, or `CALORIES_PER_GRAM`, they are available for use indirectly when the `Pizza` object needs them. The `Pizza` class cannot refer to them directly by name because they are not inherited, but they do exist. Note that a `FoodItem` object was never created or needed.

listing 7.15


```
//********************************************************************
//  Pizza.java       Author: Lewis/Loftus
//
//  Represents a pizza, which is a food item. Used to demonstrate
//  indirect referencing through inheritance.
//********************************************************************

public class Pizza extends FoodItem
{
   //-----------------------------------------------------------------
   //  Sets up a pizza with the specified amount of fat (assumes
   //  eight servings).
   //-----------------------------------------------------------------
   public Pizza (int fatGrams)
   {
      super (fatGrams, 8);
   }
}
```

Figure 7.7 lists each variable and method declared in the `FoodItem` class and indicates whether it exists in or is inherited by the `Pizza` class. Note that every `FoodItem` member exists in the `Pizza` class, no matter how it is declared. The items that are not inherited can be referenced only indirectly.

7.4 polymorphism

Usually, the type of a reference variable matches exactly the class of the object to which it refers. That is, if we declare a reference as follows, the `bishop` reference is used to refer to an object created by instantiating the `ChessPiece` class.

```
ChessPiece bishop;
```

However, the relationship between a reference variable and the object it refers to is more flexible than that.

key concept

A polymorphic reference can refer to different types of objects over time.

The term *polymorphism* can be defined as "having many forms." A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. The specific method invoked through a polymorphic reference can change from one invocation to the next.

Consider the following line of code:

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

Declared in `FoodItem` class	Defined in `Pizza` class	Inherited in `Pizza` class
`CALORIES_PER_GRAM`	yes	no, because the constant is private
`fatGrams`	yes	no, because the variable is private
`servings`	yes	yes, because the variable is protected
`FoodItem`	yes	no, because the constructors are not inherited
`calories`	yes	no, because the method is private
`caloriesPerServing`	yes	yes, because the method is public

figure 7.7 The relationship between `FoodItem` members and the `Pizza` class

If the reference `obj` is polymorphic, it can refer to different types of objects at different times. If that line of code is in a loop or in a method that is called more than once, that line of code might call a different version of the `doIt` method each time it is invoked.

At some point, the commitment is made to execute certain code to carry out a method invocation. This commitment is referred to as *binding* a method invocation to a method definition. In most situations, the binding of a method invocation to a method definition can occur at compile time. For polymorphic references, however, the decision cannot be made until run time. The method definition that is used is based on the object that is being referred to by the reference variable at that moment. This deferred commitment is called *late binding* or *dynamic binding*. It is less efficient than binding at compile time because the decision must be made during the execution of the program. This overhead is generally acceptable in light of the flexibility that a polymorphic reference provides.

We can create a polymorphic reference in Java in two ways: using inheritance and using interfaces. This section describes how we can accomplish polymorphism using inheritance. Later in the chapter we revisit the issue of interfaces and describe how polymorphism can be accomplished using interfaces as well.

references and class hierarchies

In Java, a reference that is declared to refer to an object of a particular class can also be used to refer to an object of any class related to it by inheritance. For example, if the class `Mammal` is used to derive the class `Horse`, then a `Mammal` reference can be used to refer to an object of class `Horse`. This ability is shown in the following code segment:

key concept

A reference variable can refer to any object created from any class related to it by inheritance.

```
Mammal pet;
Horse secretariat = new Horse();
pet = secretariat;  // a valid assignment
```

The reverse operation, assigning the `Mammal` object to a `Horse` reference, is also valid but requires an explicit cast. Assigning a reference in this direction is generally less useful and more likely to cause problems because although a horse has all the functionality of a mammal (because a horse *is-a* mammal), the reverse is not necessarily true.

This relationship works throughout a class hierarchy. If the `Mammal` class were derived from a class called `Animal`, the following assignment would also be valid:

```
Animal creature = new Horse();
```

Carrying this to the limit, an `Object` reference can be used to refer to any object because ultimately all classes are descendants of the `Object` class. An `ArrayList`, for example, uses polymorphism in that it is designed to hold `Object` references. That's why an `ArrayList` can be used to store any kind of object. In fact, a particular `ArrayList` can be used to hold several different types of objects at one time because, in essence, they are all `Object` objects.

polymorphism via inheritance

The reference variable `creature`, as defined in the previous section, can be polymorphic because at any point in time it can refer to an `Animal` object, a `Mammal` object, or a `Horse` object. Suppose that all three of these classes have a method called `move` that is implemented in different ways (because the child class overrode the definition it inherited). The following invocation calls the `move` method, but the particular version of the method it calls is determined at runtime:

```
creature.move();
```

key concept

A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.

At the point when this line is executed, if `creature` currently refers to an `Animal` object, the `move` method of the `Animal` class is invoked. Likewise, if `creature` currently refers to a `Mammal` or `Horse` object, the `Mammal` or `Horse` version of `move` is invoked, respectively.

Of course, since `Animal` and `Mammal` represent general concepts, they may be defined as abstract classes. This situation does not eliminate the ability to have polymorphic references. Suppose the `move` method in the `Mammal` class is abstract, and is given unique definitions in the `Horse`, `Dog`, and `Whale` classes (all derived from `Mammal`). A `Mammal` reference variable can be used to refer to any objects created from any of the `Horse`, `Dog`, and `Whale` classes, and can be used to execute the `move` method on any of them.

Let's look at another situation. Consider the class hierarchy shown in Fig. 7.8. The classes in it represent various types of employees that might be employed at a particular company. Let's explore an example that uses this hierarchy to demonstrate several inheritance issues, including polymorphism.

The `Firm` class shown in Listing 7.16 contains a `main` driver that creates a `Staff` of employees and invokes the `payday` method to pay them all. The program output includes information about each employee and how much each is paid (if anything).

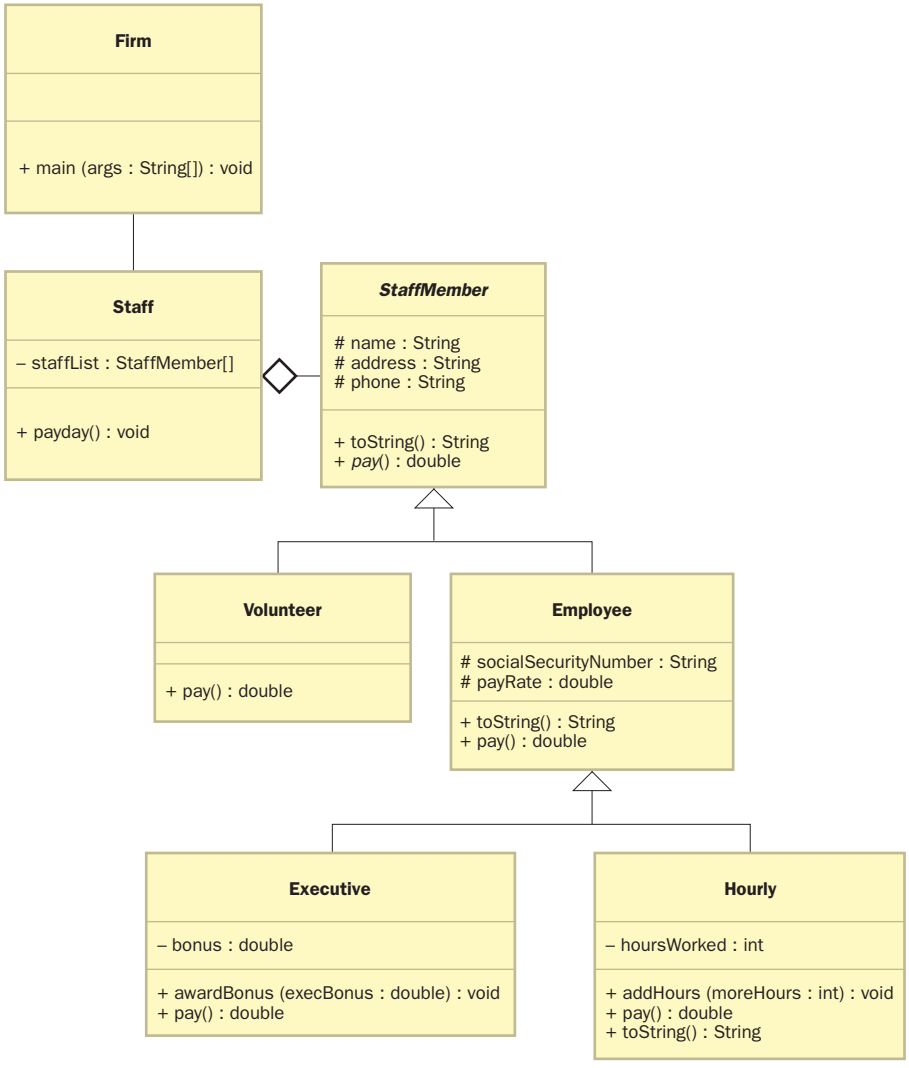


figure 7.8 A class hierarchy of employees

listing 7.16


```
//********************************************************************
//  Firm.java       Author: Lewis/Loftus
//
//  Demonstrates polymorphism via inheritance.
//********************************************************************

public class Firm
{
   //-----------------------------------------------------------------
   //  Creates a staff of employees for a firm and pays them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Staff personnel = new Staff();

      personnel.payday();
   }
}
```

output

```
Name: Sam
Address: 123 Main Line
Phone: 555-0469
Social Security Number: 123-45-6789
Paid: 2923.07
-----------------------------------
Name: Carla
Address: 456 Off Line
Phone: 555-0101
Social Security Number: 987-65-4321
Paid: 1246.15
-----------------------------------
Name: Woody
Address: 789 Off Rocker
Phone: 555-0000
Social Security Number: 010-20-3040
Paid: 1169.23
-----------------------------------
Name: Diane
Address: 678 Fifth Ave.
Phone: 555-0690
Social Security Number: 958-47-3625
Current hours: 40
Paid: 422.0
```

listing 7.16 continued

```
---------------------------------
Name: Norm
Address: 987 Suds Blvd.
Phone: 555-8374
Thanks!
---------------------------------
Name: Cliff
Address: 321 Duds Lane
Phone: 555-7282
Thanks!
---------------------------------
```

The `Staff` class shown in Listing 7.17 maintains an array of objects that represent individual employees of various kinds. Note that the array is declared to hold `StaffMember` references, but it is actually filled with objects created from several other classes, such as `Executive` and `Employee`. These classes are all descendants of the `StaffMember` class, so the assignments are valid.

The `payday` method of the `Staff` class scans through the list of employees, printing their information and invoking their `pay` methods to determine how much each employee should be paid. The invocation of the `pay` method is polymorphic because each class has its own version of the `pay` method.

The `StaffMember` class shown in Listing 7.18 is abstract. It does not represent a particular type of employee and is not intended to be instantiated. Rather, it serves as the ancestor of all employee classes and contains information that applies to all employees. Each employee has a name, address, and phone number, so variables to store these values are declared in the `StaffMember` class and are inherited by all descendants.

The `StaffMember` class contains a `toString` method to return the information managed by the `StaffMember` class. It also contains an abstract method called `pay`, which takes no parameters and returns a value of type `double`. At the generic `StaffMember` level, it would be inappropriate to give a definition for this method. The descendants of `StaffMember`, however, each provide their own

listing 7.17


```
//********************************************************************
//  Staff.java       Author: Lewis/Loftus
//
//  Represents the personnel staff of a particular business.
//********************************************************************

public class Staff
{
   private StaffMember[] staffList;

   //-----------------------------------------------------------------
   //  Sets up the list of staff members.
   //-----------------------------------------------------------------
   public Staff ()
   {
      staffList = new StaffMember[6];

      staffList[0] = new Executive ("Sam", "123 Main Line",
         "555-0469", "123-45-6789", 2423.07);

      staffList[1] = new Employee ("Carla", "456 Off Line",
         "555-0101", "987-65-4321", 1246.15);
      staffList[2] = new Employee ("Woody", "789 Off Rocker",
         "555-0000", "010-20-3040", 1169.23);

      staffList[3] = new Hourly ("Diane", "678 Fifth Ave.",
         "555-0690", "958-47-3625", 10.55);

      staffList[4] = new Volunteer ("Norm", "987 Suds Blvd.",
         "555-8374");
      staffList[5] = new Volunteer ("Cliff", "321 Duds Lane",
         "555-7282");

      ((Executive)staffList[0]).awardBonus (500.00);

      ((Hourly)staffList[3]).addHours (40);
   }

   //-----------------------------------------------------------------
   //  Pays all staff members.
   //-----------------------------------------------------------------
```

listing 7.17 continued

```
   public void payday ()
   {
      double amount;

      for (int count=0; count < staffList.length; count++)
      {
         System.out.println (staffList[count]);

         amount = staffList[count].pay();  // polymorphic

         if (amount == 0.0)
            System.out.println ("Thanks!");
         else
            System.out.println ("Paid: " + amount);

         System.out.println ("-----------------------------------");
      }
   }
}
```

specific definition for `pay`. By defining `pay` abstractly in `StaffMember`, the `payday` method of `Staff` can polymorphically pay each employee.

The `Volunteer` class shown in Listing 7.19 represents a person that is not compensated monetarily for his or her work. We keep track only of a volunteer's basic information, which is passed into the constructor of `Volunteer`, which in turn passes it to the `StaffMember` constructor using the `super` reference. The `pay` method of `Volunteer` simply returns a zero pay value. If `pay` had not been overridden, the `Volunteer` class would have been considered abstract and could not have been instantiated.

Note that when a volunteer gets "paid" in the `payday` method of `Staff`, a simple expression of thanks is printed. In all other situations, where the pay value is greater than zero, the payment itself is printed.

The `Employee` class shown in Listing 7.20 represents an employee that gets paid at a particular rate each pay period. The pay rate, as well as the employee's social security number, is passed along with the other basic information to the `Employee` constructor. The basic information is passed to the constructor of `StaffMember` using the `super` reference.

listing 7.18


```
//********************************************************************
//  StaffMember.java       Author: Lewis/Loftus
//
//  Represents a generic staff member.
//********************************************************************

abstract public class StaffMember
{
   protected String name;
   protected String address;
   protected String phone;

   //-----------------------------------------------------------------
   //  Sets up a staff member using the specified information.
   //-----------------------------------------------------------------
   public StaffMember (String eName, String eAddress, String ePhone)
   {
      name = eName;
      address = eAddress;
      phone = ePhone;
   }

   //-----------------------------------------------------------------
   //  Returns a string including the basic employee information.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = "Name: " + name + "\n";

      result += "Address: " + address + "\n";
      result += "Phone: " + phone;

      return result;
   }

   //-----------------------------------------------------------------
   //  Derived classes must define the pay method for each type of
   //  employee.
   //-----------------------------------------------------------------
   public abstract double pay();
}
```

listing 7.19


```
//********************************************************************
//  Volunteer.java       Author: Lewis/Loftus
//
//  Represents a staff member that works as a volunteer.
//********************************************************************

public class Volunteer extends StaffMember
{
   //-----------------------------------------------------------------
   //  Sets up a volunteer using the specified information.
   //-----------------------------------------------------------------
   public Volunteer (String eName, String eAddress, String ePhone)
   {
      super (eName, eAddress, ePhone);
   }

   //-----------------------------------------------------------------
   //  Returns a zero pay value for this volunteer.
   //-----------------------------------------------------------------
   public double pay()
   {
      return 0.0;
   }
}
```

The `toString` method of `Employee` is overridden to concatenate the additional information that `Employee` manages to the information returned by the parent's version of `toString`, which is called using the `super` reference. The `pay` method of an `Employee` simply returns the pay rate for that employee.

The `Executive` class shown in Listing 7.21 represents an employee that may earn a bonus in addition to his or her normal pay rate. The `Executive` class is derived from `Employee` and therefore inherits from both `StaffMember` and `Employee`. The constructor of `Executive` passes along its information to the `Employee` constructor and sets the executive bonus to zero.

listing 7.20


```
//********************************************************************
//  Employee.java       Author: Lewis/Loftus
//
//  Represents a general paid employee.
//********************************************************************

public class Employee extends StaffMember
{
   protected String socialSecurityNumber;
   protected double payRate;

   //-----------------------------------------------------------------
   //  Sets up an employee with the specified information.
   //-----------------------------------------------------------------
   public Employee (String eName, String eAddress, String ePhone,
                    String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone);

      socialSecurityNumber = socSecNumber;
      payRate = rate;
   }

   //-----------------------------------------------------------------
   //  Returns information about an employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nSocial Security Number: " + socialSecurityNumber;

      return result;
   }

   //-----------------------------------------------------------------
   //  Returns the pay rate for this employee.
   //-----------------------------------------------------------------
   public double pay()
   {
      return payRate;
   }
}
```

listing 7.21


```
//********************************************************************
//  Executive.java       Author: Lewis/Loftus
//
//  Represents an executive staff member, who can earn a bonus.
//********************************************************************

public class Executive extends Employee
{
   private double bonus;

   //-----------------------------------------------------------------
   //  Sets up an executive with the specified information.
   //-----------------------------------------------------------------
   public Executive (String eName, String eAddress, String ePhone,
                     String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone, socSecNumber, rate);

      bonus = 0;  // bonus has yet to be awarded
   }

   //-----------------------------------------------------------------
   //  Awards the specified bonus to this executive.
   //-----------------------------------------------------------------
   public void awardBonus (double execBonus)
   {
      bonus = execBonus;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the pay for an executive, which is the
   //  regular employee payment plus a one-time bonus.
   //-----------------------------------------------------------------
   public double pay()
   {
      double payment = super.pay() + bonus;

      bonus = 0;

      return payment;
   }
}
```

A bonus is awarded to an executive using the `awardBonus` method. This method is called in the `payday` method in `Staff` for the only executive that is part of the `personnel` array. Note that the generic `StaffMember` reference must be cast into an `Executive` reference to invoke the `awardBonus` method (which doesn't exist for a `StaffMember`).

The `Executive` class overrides the `pay` method so that it first determines the payment as it would for any employee, then adds the bonus. The `pay` method of the `Employee` class is invoked using `super` to obtain the normal payment amount. This technique is better than using just the `payRate` variable because if we choose to change how `Employee` objects get paid, the change will automatically be reflected in `Executive`. After the bonus is awarded, it is reset to zero.

The `Hourly` class shown in Listing 7.22 represents an employee whose pay rate is applied on an hourly basis. It keeps track of the number of hours worked in the current pay period, which can be modified by calls to the `addHours` method. This method is called from the `payday` method of `Staff`. The `pay` method of `Hourly` determines the payment based on the number of hours worked, and then resets the hours to zero.

listing 7.22


```
//********************************************************************
//  Hourly.java       Author: Lewis/Loftus
//
//  Represents an employee that gets paid by the hour.
//********************************************************************

public class Hourly extends Employee
{
   private int hoursWorked;

   //-----------------------------------------------------------------
   //  Sets up this hourly employee using the specified information.
   //-----------------------------------------------------------------
   public Hourly (String eName, String eAddress, String ePhone,
                  String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone, socSecNumber, rate);

      hoursWorked = 0;
   }
```

listing 7.22 continued

```
   //-----------------------------------------------------------------
   //  Adds the specified number of hours to this employee's
   //  accumulated hours.
   //-----------------------------------------------------------------
   public void addHours (int moreHours)
   {
      hoursWorked += moreHours;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the pay for this hourly employee.
   //-----------------------------------------------------------------
   public double pay()
   {
      double payment = payRate * hoursWorked;

      hoursWorked = 0;

      return payment;
   }

   //-----------------------------------------------------------------
   //  Returns information about this hourly employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nCurrent hours: " + hoursWorked;

      return result;
   }
}
```

7.5 interfaces revisited

We introduced interfaces in Chapter 5. We revisit them here because they have a lot in common with the topic of inheritance. Like classes, interfaces can be organized into inheritance hierarchies. And just as we can accomplish polymorphism using the inheritance relationship, we can also accomplish it using interfaces. This section discusses both of these topics.

interface hierarchies

The concept of inheritance can be applied to interfaces as well as classes. That is, one interface can be derived from another interface. These relationships can form an *interface hierarchy,* which is similar to a class hierarchy. Inheritance relationships between interfaces are shown in UML using the same connection (an arrow with an open arrowhead) as they are with classes.

key concept
Inheritance can be applied to interfaces so that one interface can be derived from another.

When a parent interface is used to derive a child interface, the child inherits all abstract methods and constants of the parent. Any class that implements the child interface must implement all of the methods. There are no restrictions on the inheritance between interfaces, as there are with protected and private members of a class, because all members of an interface are public.

Class hierarchies and interface hierarchies do not overlap. That is, an interface cannot be used to derive a class, and a class cannot be used to derive an interface. A class and an interface interact only when a class is designed to implement a particular interface.

polymorphism via interfaces

key concept
An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements that interface.

As we've seen many times, a class name is used to declare the type of an object reference variable. Similarly, an interface name can be used as the type of a reference variable as well. An interface reference variable can be used to refer to any object of any class that implements that interface.

Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
   public void speak();
   public void announce (String str);
}
```

The interface name, `Speaker`, can now be used to declare an object reference variable:

```
Speaker current;
```

The reference variable `current` can be used to refer to any object of any class that implements the `Speaker` interface. For example, if we define a class called `Philosopher` such that it implements the `Speaker` interface, we can then assign a `Philosopher` object to a `Speaker` reference as follows:

```
current = new Philosopher();
```

This assignment is valid because a `Philosopher` is, in fact, a `Speaker`.

The flexibility of an interface reference allows us to create polymorphic references. As we saw earlier in this chapter, using inheritance, we can create a polymorphic reference that can refer to any one of a set of objects related by inheritance. Using interfaces, we can create similar polymorphic references, except that the objects being referenced, instead of being related by inheritance, are related by implementing the same interface.

key concept

Interfaces allow us to make polymorphic references in which the method that is invoked is based on the particular object being referenced at the time.

For example, if we create a class called `Dog` that also implements the `Speaker` interface, it can be assigned to a `Speaker` reference variable. The same reference, in fact, can at one point refer to a `Philosopher` object and then later refer to a `Dog` object. The following lines of code illustrate this:

```
Speaker guest;
guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```

In this code, the first time the `speak` method is called, it invokes the `speak` method defined in the `Philosopher` class. The second time it is called, it invokes the `speak` method of the `Dog` class. As with polymorphic references via inheritance, it is not the type of the reference that determines which method gets invoked; it depends on the type of the object that the reference points to at the moment of invocation.

Note that when we are using an interface reference variable, we can invoke only the methods defined in the interface, even if the object it refers to has other methods to which it can respond. For example, suppose the `Philosopher` class also defined a public method called `pontificate`. The second line of the following code would generate a compiler error, even though the object can in fact respond to the `pontificate` method:

```
Speaker special = new Philosopher();
special.pontificate();  // generates a compiler error
```

The problem is that the compiler can determine only that the object is a `Speaker`, and therefore can guarantee only that the object can respond to the `speak` and `announce` methods. Because the reference variable `special` could refer to a `Dog` object (which cannot pontificate), it does not allow the reference. If we know in a particular situation that such an invocation is valid, we can cast the object into the appropriate reference so that the compiler will accept it as follows:

```
((Philosopher)special).pontificate();
```

Similar to polymorphic references based in inheritance, an interface name can be used as the type of a method parameter. In such situations, any object of any class that implements the interface can be passed into the method. For example, the following method takes a `Speaker` object as a parameter. Therefore both a `Dog` object and a `Philosopher` object can be passed into it in separate invocations:

```
public void sayIt (Speaker current)
{
   current.speak();
}
```

7.6 inheritance and GUIs

The concept of inheritance affects our use of graphics and GUIs. This section explores some of these issues.

It's important in these discussions to recall that there are two primary GUI APIs used in Java: the Abstract Windowing Toolkit (AWT) and the Swing classes. The AWT is the original set of graphics classes in Java. Swing classes were introduced later, adding components that provided much more functionality than their AWT counterparts. In general, we use Swing components in our examples in this book.

applets revisited

In previous chapters, we've created applets using inheritance. Initially, we extended the `Applet` class, which is an original AWT component that is part of the `java.applet` package. In Chapter 5 and beyond, we've derived our applets from the `JApplet` class, which is the Swing version. The primary difference between these two classes is that a `JApplet` has a content pane to which GUI components are added. Also, in general, a `JApplet` component should not be drawn on directly. It's better to draw on a panel and add that panel to the applet to be displayed, especially if there is to be user interaction.

key concept

An applet is a good example of inheritance. The `JApplet` parent class handles characteristics common to all applets.

The extension of an applet class demonstrates a classic use of inheritance, allowing the parent class to shoulder the responsibilities that apply to all of its descendants. The `JApplet` class is already designed to handle all of the details concerning applet creation and execution. For example, an applet program interacts with a browser, can accept parameters through HTML code, and is constrained by certain security limitations. The `JApplet` class already takes care of these details in a generic way that applies to all applets.

Because of inheritance, the applet class that we write (the one derived from `JApplet`) is ready to focus on the purpose of that particular program. In other words, the only issues that we address in our applet code are those that make it different from other applets.

Note that we've been using applets even before we examined what inheritance accomplishes for us and what the parent applet class does in particular. We used the parent applet classes simply for the services it provides. Therefore applets are another wonderful example of abstraction in which certain details can be ignored.

the component class hierarchy

All of the Java classes that define GUI components are part of a class hierarchy, shown in part in Fig. 7.9. Almost all Swing GUI components are derived from

key concept
The classes that represent Java GUI components are organized into a class hierarchy.

the `JComponent` class, which defines how all components work in general. `JComponent` is derived from the `Container` class, which in turn is derived from the `Component` class.

Both `Container` and `Component` are original AWT classes. The `Component` class contains much of the general functionality that applies to all GUI components, such as basic painting and event handling. So although we may prefer to use some of the specific Swing components, they are based on core AWT concepts and respond to the same events as AWT components. Because they are derived from `Container`, many Swing components can serve as containers, though in most circumstances those abilities are curtailed. For example, a `JLabel` object can contain an image (as described in the next chapter) but it cannot be used as a generic container to which any component can be added.

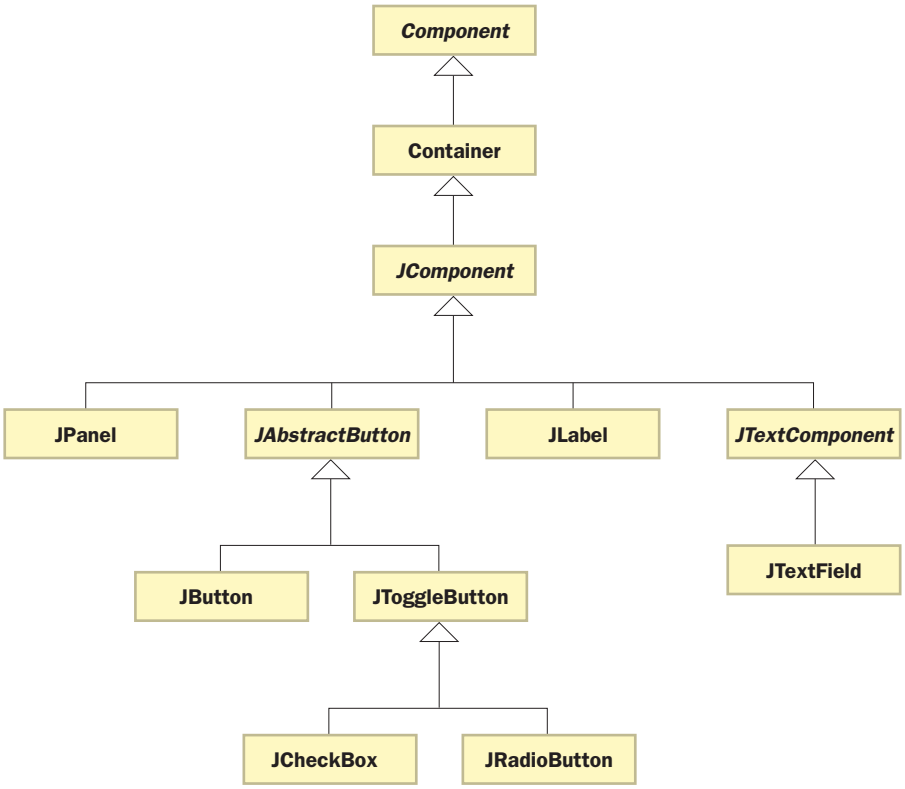


figure 7.9 Part of the GUI component class hierarchy

Many features that apply to all Swing components are defined in the `JComponent` class and are inherited into its descendants. For example, we have the ability to put a border on any Swing component (we discuss this more in Chapter 9). This ability is defined, only once, in the `JComponent` class and is inherited by any class that is derived, directly or indirectly, from it.

Some component classes, such as `JPanel` and `JLabel`, are derived directly from `JComponent`. Other component classes are nested further down in the inheritance hierarchy structure. For example, the `JAbstractButton` class is an abstract class that defines the functionality that applies to all types of GUI buttons. `JButton` is derived directly from it. However, note that `JCheckBox` and `JRadioButton` are both derived from a class called `JToggleButton`, which embodies the common characteristics for buttons that can be in one of two states. The set of classes that define GUI buttons shows once again how common characteristics are put at appropriately high levels of the class hierarchy rather than duplicated in multiple classes.

The world of text components demonstrates this as well. The `JTextField` class that we've used in previous examples is one of many Java GUI components that support the management of text data. They are organized under a class called `JTextComponent`. Keep in mind that there are many GUI component classes that are not shown in the diagram in Fig. 7.9.

Painting is another GUI feature affected by the inheritance hierarchy. The `paint` method we've used in applets is defined in the `Component` class. The `Applet` class inherits the default version of this method, which we have regularly overridden in our applet programs to paint particular shapes. Most Swing classes, however, use a method called `paintComponent` to perform custom painting. Usually, we will draw on a `JPanel` using its `paintComponent` method and use the `super` reference to invoke the version of the `paintComponent` method defined in `JComponent`, which draws the background and outline of the component. This technique is demonstrated in the next section.

7.7 mouse events

Let's examine the events that are generated when using a mouse. Java divides these events into two categories: *mouse events* and *mouse motion events*. The table in Fig. 7.10 defines these events.

Mouse Event	Description
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 Motion Event	Description
mouse moved	The mouse is moved.
mouse dragged	The mouse is moved while the mouse button is pressed down.

figure 7.10 Mouse events and mouse motion events

When you click the mouse button over a Java GUI component, three events are generated: one when the mouse button is pushed down (*mouse pressed*) and two when it is let up (*mouse released* and *mouse clicked*). A mouse click is defined as pressing and releasing the mouse button in the same location. If you press the mouse button down, move the mouse, and then release the mouse button, a mouse clicked event is not generated.

A component will generate a *mouse entered* event when the mouse pointer passes into its graphical space. Likewise, it generates a *mouse exited* event when the mouse pointer leaves.

key concept

Moving the mouse and clicking the mouse button generate mouse events to which a program can respond.

Mouse motion events, as the name implies, occur while the mouse is in motion. The *mouse moved* event indicates simply that the mouse is in motion. The *mouse dragged* event is generated when the user has pressed the mouse button down and moved the mouse without releasing the button. Mouse motion events are generated many times, very quickly, while the mouse is in motion.

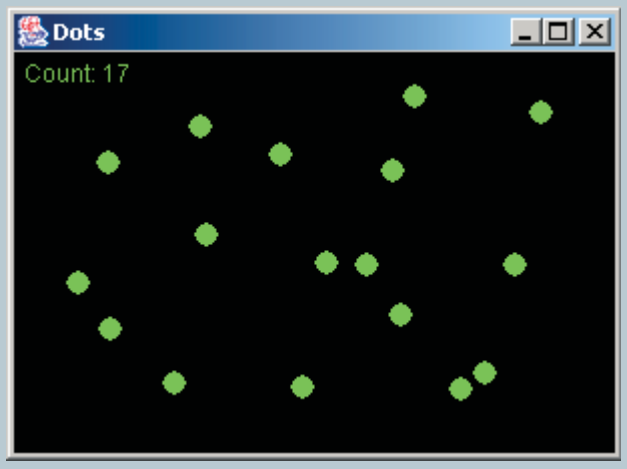
In a specific situation, we may care about only one or two mouse events. What we listen for depends on what we are trying to accomplish.

The `Dots` program shown in Listing 7.23 responds to one mouse event. Specifically, it draws a green dot at the location of the mouse pointer whenever the mouse button is pressed.

listing 7.23

```
//********************************************************************
//  Dots.java       Author: Lewis/Loftus
//
//  Demonstrates mouse events and drawing on a panel.
//********************************************************************

import javax.swing.*;

public class Dots
{
   //-----------------------------------------------------------------
   //  Creates and displays the application frame.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame dotsFrame = new JFrame ("Dots");
      dotsFrame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

      dotsFrame.getContentPane().add (new DotsPanel());

      dotsFrame.pack();
      dotsFrame.show();
   }
}
```

display



The `main` method of the `Dots` class creates a frame and adds one panel to it. That panel is defined by the `DotsPanel` class shown in Listing 7.24. The `DotsPanel` class is derived from `JPanel`. This panel serves as the surface on which the dots are drawn.

listing 7.24

```
//********************************************************************
//  DotsPanel.java       Author: Lewis/Loftus
//
//  Represents the primary panel for the Dots program on which the
//  dots are drawn.
//********************************************************************

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class DotsPanel extends JPanel
{
   private final int WIDTH = 300, HEIGHT = 200;
   private final int RADIUS = 6;

   private ArrayList pointList;
   private int count;

   //-----------------------------------------------------------------
   //  Sets up this panel to listen for mouse events.
   //-----------------------------------------------------------------
   public DotsPanel()
   {
      pointList = new ArrayList();
      count = 0;

      addMouseListener (new DotsListener());

      setBackground (Color.black);
      setPreferredSize (new Dimension(WIDTH, HEIGHT));
   }

   //-----------------------------------------------------------------
   //  Draws all of the dots stored in the list.
   //-----------------------------------------------------------------
```

listing 7.24 continued

```
   public void paintComponent (Graphics page)
   {
      super.paintComponent(page);

      page.setColor (Color.green);

      // Retrieve an iterator for the ArrayList of points
      Iterator pointIterator = pointList.iterator();

      while (pointIterator.hasNext())
      {
         Point drawPoint = (Point) pointIterator.next();
         page.fillOval (drawPoint.x - RADIUS, drawPoint.y - RADIUS,
                        RADIUS * 2, RADIUS * 2);
      }

      page.drawString ("Count: " + count, 5, 15);
   }

   //*****************************************************************
   //  Represents the listener for mouse events.
   //*****************************************************************
   private class DotsListener implements MouseListener
   {
      //--------------------------------------------------------------
      //  Adds the current point to the list of points and redraws
      //  whenever the mouse button is pressed.
      //--------------------------------------------------------------
      public void mousePressed (MouseEvent event)
      {
         pointList.add (event.getPoint());
         count++;
         repaint();
      }

      //--------------------------------------------------------------
      //  Provide empty definitions for unused event methods.
      //--------------------------------------------------------------
      public void mouseClicked (MouseEvent event) {}
      public void mouseReleased (MouseEvent event) {}
      public void mouseEntered (MouseEvent event) {}
      public void mouseExited (MouseEvent event) {}
   }
}
```

The DotsPanel class keeps track of a list of Point objects that represent all of the locations at which the user has clicked the mouse. A Point class represents the (*x*, *y*) coordinates of a given point in two-dimensional space. It provides public access to the instance variables x and y for the point. Each time the panel is painted, all of the points stored in the list are drawn. The list is maintained as an ArrayList object. To draw the points, an Iterator object is obtained from the ArrayList so that each point can be processed in turn. We discussed the ArrayList class in Chapter 6 and the Iterator interface in Chapter 5.

The listener for the mouse pressed event is defined as a private inner class that implements the MouseListener interface. The mousePressed method is invoked by the panel each time the user presses down on the mouse button while it is over the panel.

A mouse event always occurs at some point in space, and the object that represents that event keeps track of that location. In a mouse listener, we can get and use that point whenever we need it. In the Dots program, each time the mousePressed method is called, the location of the event is obtained using the getPoint method of the MouseEvent object. That point is stored in the ArrayList, and the panel is then repainted.

Note that, unlike the ActionListener and ItemListener interfaces that we've used in previous examples, which contain one method each, the MouseListener interface contains five methods. For this program, the only event in which we are interested is the mouse pressed event. Therefore, the only method in which we have any interest is the mousePressed method. However, implementing an interface means we must provide definitions for all methods in the interface. Therefore we provide empty methods corresponding to the other events. When those events are generated, the empty methods are called, but no code is executed.

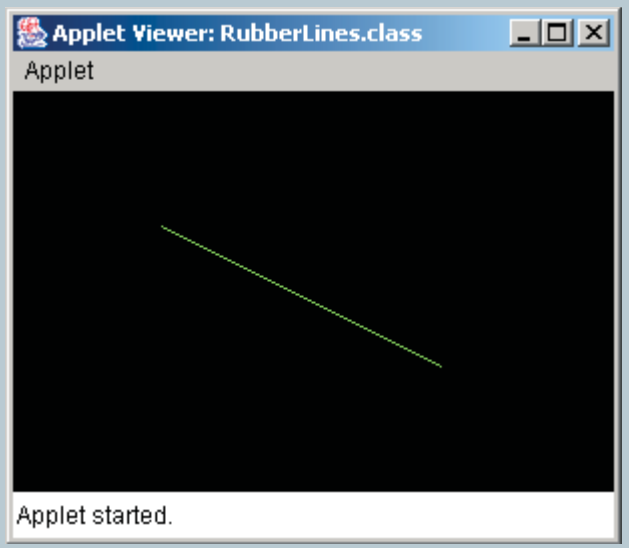
Let's look at an example that responds to two mouse-oriented events. The RubberLines program shown in Listing 7.25 draws a line between two points. The first point is determined by the location at which the mouse is first pressed down. The second point changes as the mouse is dragged while the mouse button is held down. When the button is released, the line remains fixed between the first and second points. When the mouse button is pressed again, a new line is started. This program is implemented as an applet.

The panel on which the lines are drawn is represented by the RubberLinesPanel class shown in Listing 7.26. Because we need to listen for both a mouse pressed event and a mouse dragged event, we need a listener that responds to both mouse events and mouse motion events. Note that the listener class implements both the MouseListener and MouseMotionListener inter-

listing 7.25

```
//********************************************************************
//  RubberLines.java       Author: Lewis/Loftus
//
//  Demonstrates mouse events and rubberbanding.
//********************************************************************

import javax.swing.*;

public class RubberLines extends JApplet
{
   private final int WIDTH = 300, HEIGHT = 200;

   //-----------------------------------------------------------------
   //  Sets up the applet to contain the drawing panel.
   //-----------------------------------------------------------------
   public void init()
   {
      getContentPane().add (new RubberLinesPanel());

      setSize (WIDTH, HEIGHT);
   }
}
```

display



faces. It must therefore implement all methods of both classes. The two methods of interest, `mousePressed` and `mouseDragged`, are implemented, and the rest are given empty definitions.

listing 7.26

```
//********************************************************************
//  RubberLinesPanel.java       Author: Lewis/Loftus
//
//  Represents the primary drawing panel for the RubberLines applet.
//********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RubberLinesPanel extends JPanel
{
   private Point point1 = null, point2 = null;

   //-----------------------------------------------------------------
   //  Sets up the applet to listen for mouse events.
   //-----------------------------------------------------------------
   public RubberLinesPanel()
   {
      LineListener listener = new LineListener();
      addMouseListener (listener);
      addMouseMotionListener (listener);

      setBackground (Color.black);
   }

   //-----------------------------------------------------------------
   //  Draws the current line from the intial mouse down point to
   //  the current position of the mouse.
   //-----------------------------------------------------------------
   public void paintComponent (Graphics page)
   {
      super.paintComponent (page);

      page.setColor (Color.green);
      if (point1 != null && point2 != null)
         page.drawLine (point1.x, point1.y, point2.x, point2.y);
   }
```

listing 7.26 continued

```
   //*****************************************************************
   //  Represents the listener for all mouse events.
   //*****************************************************************
   private class LineListener implements MouseListener,
                                         MouseMotionListener
   {
      //--------------------------------------------------------------
      //  Captures the initial position at which the mouse button is
      //  pressed.
      //--------------------------------------------------------------
      public void mousePressed (MouseEvent event)
      {
         point1 = event.getPoint();
      }

      //--------------------------------------------------------------
      //  Gets the current position of the mouse as it is dragged and
      //  draws the line to create the rubberband effect.
      //--------------------------------------------------------------
      public void mouseDragged (MouseEvent event)
      {
         point2 = event.getPoint();
         repaint();
      }

      //--------------------------------------------------------------
      //  Provide empty definitions for unused event methods.
      //--------------------------------------------------------------
      public void mouseClicked (MouseEvent event) {}
      public void mouseReleased (MouseEvent event) {}
      public void mouseEntered (MouseEvent event) {}
      public void mouseExited (MouseEvent event) {}
      public void mouseMoved (MouseEvent event) {}
   }
}
```

key concept
Rubberbanding is the visual effect created when a graphical shape seems to expand and contract as the mouse is dragged.

When the `mousePressed` method is called, the variable `point1` is set. Then, as the mouse is dragged, the variable `point2` is continually reset and the panel repainted. Therefore the line is constantly being redrawn as the mouse is dragged, giving the appearance that one line is being stretched between a fixed point and a moving point. This effect is called *rubberbanding* and is common in graphical programs.

Note that, in the `RubberLinesPanel` constructor, the listener object is added to the panel twice: once as a mouse listener and once as a mouse motion listener. The method called to add the listener must correspond to the object passed as the parameter. In this case, we had one object that served as a listener for both categories of events. We could have had two listener classes if desired: one listening for mouse events and one listening for mouse motion events. A component can have multiple listeners for various event categories.

Also note that this program draws one line at a time. That is, when the user begins to draw another line with a new mouse click, the previous one disappears. This is because the `paintComponent` method redraws its background, eliminating the line every time. To see the previous lines, we'd have to keep track of them, perhaps using an `ArrayList` as was done in the `Dots` program. This modification to the `RubberLines` program is left as a programming project.

extending event adapter classes

In previous event-based examples, we've created the listener classes by implementing a particular listener interface. For instance, to create a class that listens for mouse events, we created a listener class that implements the `MouseListener` interface. As we saw in the `Dots` and `RubberLines` programs, a listener interface often contains event methods that are not important to a particular program, in which case we provided empty definitions to satisfy the interface requirement.

key concept
A listener class can be created by deriving it from an event adapter class.

An alternative technique for creating a listener class is to extend an *event adapter class*. Each listener interface that contains more than one method has a corresponding adapter class that already contains empty definitions for all of the methods in the interface. To create a listener, we can derive a new listener class from the appropriate adapter class and override any event methods in which we are interested. Using this technique, we no longer must provide empty definitions for unused methods.

The program shown in Listing 7.27 is an applet that responds to mouse click events. Whenever the mouse button is clicked over the applet, a line is drawn from the location of the mouse pointer to the center of the applet. The distance that line represents in pixels is displayed.

The structure of the `OffCenter` program is similar to that of the `RubberLines` program. It loads a display panel, represented by the `OffCenterPanel` class shown in Listing 7.28 into the applet window.

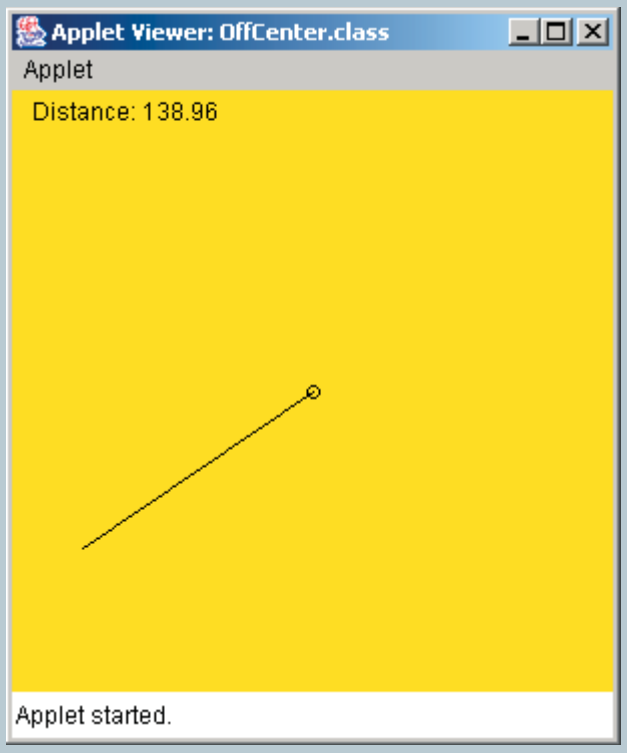
The listener class, instead of implementing the `MouseListener` interface directly as we have done in previous examples, extends the `MouseAdapter` class, which is defined in the `java.awt.event` package of the Java standard class library. The `MouseAdapter` class implements the `MouseListener` interface and contains empty definitions for all of the mouse event methods. In our listener class, we override the definition of the `mouseClicked` method to suit our needs.

listing 7.27

```
//********************************************************************
//  OffCenter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an event adatpter class.
//********************************************************************

import javax.swing.*;

public class OffCenter extends JApplet
{
   private final int WIDTH = 300, HEIGHT = 300;

   //-----------------------------------------------------------------
   //  Sets up the applet.
   //-----------------------------------------------------------------
   public void init()
   {
      getContentPane().add(new OffCenterPanel (WIDTH, HEIGHT));

      setSize (WIDTH, HEIGHT);
   }
}
```

listing 7.27 continued

display



Because we inherit the other empty methods corresponding to the rest of the mouse events, we don't have to provide our own empty definitions.

Because of inheritance, we now have a choice when it comes to creating event listeners. We can implement an event listener interface, or we can extend an event adapter class. This is a design decision that should be considered carefully. The best technique depends on the situation.

listing 7.28

```
//********************************************************************
//  OffCenterPanel.java       Author: Lewis/Loftus
//
//  Represents the primary drawing panel for the OffCenter applet.
//********************************************************************

import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;
import javax.swing.*;

public class OffCenterPanel extends JPanel
{
   private DecimalFormat fmt;
   private Point current;
   private int centerX, centerY;
   private double length;

   //-----------------------------------------------------------------
   //  Sets up the panel and necessary data.
   //-----------------------------------------------------------------
   public OffCenterPanel (int width, int height)
   {
      addMouseListener (new OffCenterListener());

      centerX = width / 2;
      centerY = height / 2;

      fmt = new DecimalFormat ("0.##");

      setBackground (Color.yellow);
   }

   //-----------------------------------------------------------------
   //  Draws a line from the mouse pointer to the center point of
   //  the applet and displays the distance.
   //-----------------------------------------------------------------
   public void paintComponent (Graphics page)
   {
      super.paintComponent (page);

      page.setColor (Color.black);
      page.drawOval (centerX-3, centerY-3, 6, 6);
```

listing 7.28 continued

```
      if (current != null)
      {
         page.drawLine (current.x, current.y, centerX, centerY);
         page.drawString ("Distance: " + fmt.format(length), 10, 15);
      }
   }

   //*****************************************************************
   //  Represents the listener for mouse events.
   //*****************************************************************
   private class OffCenterListener extends MouseAdapter
   {
      //--------------------------------------------------------------
      //  Computes the distance from the mouse pointer to the center
      //  point of the applet.
      //--------------------------------------------------------------
      public void mouseClicked (MouseEvent event)
      {
         current = event.getPoint();
         length = Math.sqrt(Math.pow((current.x-centerX), 2) +
                            Math.pow((current.y-centerY), 2));
         repaint();
      }
   }
}
```

summary of key concepts

- Inheritance is the process of deriving a new class from an existing one.
- One purpose of inheritance is to reuse existing software.
- Inherited variables and methods can be used in the derived class as if they had been declared locally.
- Inheritance creates an is-a relationship between all parent and child classes.
- Visibility modifiers determine which variables and methods are inherited. Protected visibility provides the best possible encapsulation that permits inheritance.
- A parent's constructor can be invoked using the `super` reference.
- A child class can override (redefine) the parent's definition of an inherited method.
- The child of one class can be the parent of one or more other classes, creating a class hierarchy.
- Common features should be located as high in a class hierarchy as is reasonably possible, minimizing maintenance efforts.
- All Java classes are derived, directly or indirectly, from the `Object` class.
- The `toString` and `equals` methods are defined in the `Object` class and therefore are inherited by every class in every Java program.
- An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.
- A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.
- All members of a superclass exist for a subclass, but they are not necessarily inherited. Only inherited members can be referenced by name in the subclass.
- A polymorphic reference can refer to different types of objects over time.
- A reference variable can refer to any object created from any class related to it by inheritance.
- A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.
- Inheritance can be applied to interfaces so that one interface can be derived from another.

- An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements that interface.
- Interfaces allow us to make polymorphic references in which the method that is invoked is based on the particular object being referenced at the time.
- An applet is a good example of inheritance. The `JApplet` parent class handles characteristics common to all applets.
- The classes that represent Java GUI components are organized into a class hierarchy.
- Moving the mouse and clicking the mouse button generate mouse events to which a program can respond.
- Rubberbanding is the visual effect created when a graphical shape seems to expand and contract as the mouse is dragged.
- A listener class can be created by deriving it from an event adapter class.

self-review questions

7.1 Describe the relationship between a parent class and a child class.

7.2 How does inheritance support software reuse?

7.3 What relationship should every class derivation represent?

7.4 Why would a child class override one or more of the methods of its parent class?

7.5 Why is the `super` reference important to a child class?

7.6 What is the significance of the `Object` class?

7.7 What is the role of an abstract class?

7.8 Are all members of a parent class inherited by the child? Explain.

7.9 What is polymorphism?

7.10 How does inheritance support polymorphism?

7.11 How is overriding related to polymorphism?

7.12 What is an interface hierarchy?

7.13 How can polymorphism be accomplished using interfaces?

7.14 What is an adapter class?

exercises

7.1 Draw a UML class diagram showing an inheritance hierarchy containing classes that represent different types of clocks. Show the variables and method names for two of these classes.

7.2 Show an alternative diagram for the hierarchy in Exercise 7.1. Explain why it may be a better or worse approach than the original.

7.3 Draw and annotate a class hierarchy that represents various types of faculty at a university. Show what characteristics would be represented in the various classes of the hierarchy. Explain how polymorphism could play a role in the process of assigning courses to each faculty member.

7.4 Experiment with a simple derivation relationship between two classes. Put `println` statements in constructors of both the parent and child classes. Do not explicitly call the constructor of the parent in the child. What happens? Why? Change the child's constructor to explicitly call the constructor of the parent. Now what happens?

7.5 What would happen if the `pay` method were not defined as an abstract method in the `StaffMember` class of the `Firm` program?

7.6 What would happen if, in the `Dots` program, we did not provide empty definitions for one or more of the unused mouse events?

7.7 The `Dots` program listens for a mouse pressed event to draw a dot. How would the program behave differently if it listened for a mouse released event instead? A mouse clicked event?

7.8 What would happen if the call to `super.paintComponent` were removed from the `paintComponent` method of the `DotsPanel` class? Remove it and run the program to test your answer.

7.9 What would happen if the call to `super.paintComponent` were removed from the `paintComponent` method of the `RubberLinesPanel` class? Remove it and run the program to test your answer. In what ways is the answer different from the answer to Exercise 7.8?

7.10 Explain how a call to the `addMouseListener` method represents a polymorphic situation.

programming projects

7.1 Design and implement a class called `MonetaryCoin` that is derived from the `Coin` class presented in Chapter 4. Store a value in the monetary coin that represents its value and add a method that returns its value. Create a `main` driver class to instantiate and compute the sum of several `MonetaryCoin` objects. Demonstrate that a monetary coin inherits its parent's ability to be flipped.

7.2 Design and implement a set of classes that define the employees of a hospital: doctor, nurse, administrator, surgeon, receptionist, janitor, and so on. Include methods in each class that are named according to the services provided by that person and that print an appropriate message. Create a `main` driver class to instantiate and exercise several of the classes.

7.3 Design and implement a set of classes that define various types of reading material: books, novels, magazines, technical journals, textbooks, and so on. Include data values that describe various attributes of the material, such as the number of pages and the names of the primary characters. Include methods that are named appropriately for each class and that print an appropriate message. Create a `main` driver class to instantiate and exercise several of the classes.

7.4 Design and implement a set of classes that keeps track of various sports statistics. Have each low-level class represent a specific sport. Tailor the services of the classes to the sport in question, and move common attributes to the higher-level classes as appropriate. Create a `main` driver class to instantiate and exercise several of the classes.

7.5 Design and implement a set of classes that keeps track of demographic information about a set of people, such as age, nationality, occupation, income, and so on. Design each class to focus on a particular aspect of data collection. Create a `main` driver class to instantiate and exercise several of the classes.

7.6 Modify the `StyleOptions` program from Chapter 6 so that it accomplishes the same task but derives its primary panel using inheritance. Specifically, replace the `StyleGUI` class with one called `StylePanel` that extends the `JPanel` class. Eliminate the `getPanel` method.

7.7 Perform the same modifications described in Programming Project 7.6 to the `QuoteOptions` program from Chapter 6.

7.8 Design and implement an application that draws a traffic light and uses a push button to change the state of the light. Derive the drawing surface from the `JPanel` class and use another panel to organize the drawing surface and the button.

7.9 Modify the `RubberLines` program from this chapter so that it shows all of the lines drawn. Show only the final lines (from initial mouse press to mouse release), not the intermediate lines drawn to show the rubberbanding effect. *Hint*: Keep track of a list of objects that represent the lines similar to how the `Dots` program kept track of multiple dots.

7.10 Design and implement an applet that counts the number of times the mouse has been clicked. Display that number in the center of the applet window.

7.11 Design and implement an application that creates a polyline shape dynamically using mouse clicks. Each mouse click adds a new line segment from the previous point. Include a button below the drawing area to clear the current polyline and begin another.

7.12 Design and implement an application that draws a circle using a rubberbanding technique. The circle size is determined by a mouse drag. Use the original mouse click location as a fixed center point. Compute the distance between the current location of the mouse pointer and the center point to determine the current radius of the circle.

7.13 Design and implement an application that serves as a mouse odometer, continually displaying how far, in pixels, the mouse has moved (while it is over the program window). Display the current odometer value using a label. *Hint*: Use the mouse movement event to determine the current position, and compare it to the last position of the mouse. Use the distance formula to see how far the mouse has traveled, and add that to a running total distance.

7.14 Design and implement an applet whose background changes color depending on where the mouse pointer is located. If the mouse pointer is on the left half of the applet window, display red; if it is on the right half, display green.

7.15 Design and implement a class that represents a spaceship, which can be drawn (side view) in any particular location. Create an applet that displays the spaceship so that it follows the movement of the mouse. When the mouse button is pressed down, have a laser beam

shoot out of the front of the spaceship (one continuous beam, not a moving projectile) until the mouse button is released.

answers to self-review questions

7.1 A child class is derived from a parent class using inheritance. The methods and variables of the parent class automatically become a part of the child class, subject to the rules of the visibility modifiers used to declare them.

7.2 Because a new class can be derived from an existing class, the characteristics of the parent class can be reused without the error-prone process of copying and modifying code.

7.3 Each inheritance derivation should represent an is-a relationship: the child *is-a* more specific version of the parent. If this relationship does not hold, then inheritance is being used improperly.

7.4 A child class may prefer its own definition of a method in favor of the definition provided for it by its parent. In this case, the child overrides (redefines) the parent's definition with its own.

7.5 The `super` reference can be used to call the parent's constructor, which cannot be invoked directly by name. It can also be used to invoke the parent's version of an overridden method.

7.6 All classes in Java are derived, directly or indirectly, from the `Object` class. Therefore all public methods of the `Object` class, such as `equals` and `toString`, are available to every object.

7.7 An abstract class is a representation of a general concept. Common characteristics and method signatures can be defined in an abstract class so that they are inherited by child classes derived from it.

7.8 A class member is not inherited if it has private visibility, meaning that it cannot be referenced by name in the child class. However, such members do exist for the child and can be referenced indirectly.

7.9 Polymorphism is the ability of a reference variable to refer to objects of various types at different times. A method invoked through such a reference is bound to different method definitions at different times, depending on the type of the object referenced.

7.10 In Java, a reference variable declared using a parent class can be used to refer to an object of the child class. If both classes contain a

method with the same signature, the parent reference can be polymorphic.

7.11 When a child class overrides the definition of a parent's method, two versions of that method exist. If a polymorphic reference is used to invoke the method, the version of the method that is invoked is determined by the type of the object being referred to, not by the type of the reference variable.

7.12 A new interface can be derived from an existing interface using inheritance, just as a new class can be derived from an existing class.

7.13 An interface name can be used as the type of a reference. Such a reference variable can refer to any object of any class that implements that interface. Because all classes implement the same interface, they have methods with common signatures, which can be dynamically bound.

7.14 An adapter class is a class that implements a listener interface, providing empty definitions for all of its methods. A listener class can be created by extending the appropriate adapter class and overriding the methods of interest.