CHAPTER
13

Software Reuse with Extended Classes

It is indeed a desirable thing to be well descended, but the glory belongs to our ancestors.

PLUTARCH
Morals

Often you will find yourself with a class that is *nearly* what you require, but not quite. Perhaps the class that you require needs alterations to one or two of the methods of an existing class, or maybe an existing class just needs a few extra methods. Object-oriented languages provide support for this situation by allowing programmers to easily create new classes that acquire most of their properties from an existing class. The original class is called the **superclass** (or *parent* class, or *ancestor* class, or *base class*—the lingo isn't entirely stabilized), and the new, slightly different class is the **extended class** (or *derived* class, or *child* class, or *descendant* class, or *subclass*).

The first section of this chapter provides an introduction to extended classes. The next two sections show two detailed programming examples, including an illustration of how many of our previous ADTs might have benefited from using an extended class.

13.1 EXTENDED CLASSES

One of the exercises in Chapter 2 was a `Clock` class to keep track of one instance of a time value such as 9:48 P.M. (see Self-Test Exercise 5 on page 48). One possible specification for this class is shown in Figure 13.1—we're not concerned about the `Clock` implementation right now. Suppose you're writing a program with various kinds of clocks: 12-hour clocks, 24-hour clocks, alarm clocks, grandfather clocks, cuckoo clocks, maybe even a computer clock. Each of these things is a `Clock`, but each of these also has additional properties that don't apply to clocks in general. For example, a `CuckooClock` might have an extra method, `isCuckooing`, that returns `true` if its cuckoo bird is currently making noise. How would you implement a `CuckooClock`, which is a `Clock` with one extra `isCuckooing` method?

One possible solution uses no new ideas: Make a copy of the original `Clock.java` file, change the name of the class to `CuckooClock`, and add an extra method, `isCuckooing`. Can you think of some potential problems with this solution? We'll end up writing a separate class declaration for each different type of clock. Even though all of these clocks have similar or identical constructors and methods, we'll still end up repeating the method implementations for each different kind of clock.

FIGURE 13.1 Specification for the Clock Class

Class Clock

❖ **public class Clock from the package edu.colorado.simulations**

A `Clock` object holds one instance of a time value such as 9:48 P.M.

Specification

◆ **Constructor for the Clock**

```
public Clock( )
```

Construct a `Clock` that is initially set to midnight.

Postcondition:

This `Clock` has been initialized with an initial time of midnight.

(continued)

(FIGURE 13.1 continued)

◆ **advance**

`public void advance(int minutes)`

Move this `Clock`'s time by a given number of minutes.

Parameters:

`minutes` – the amount to move this `Clock`'s time

Postcondition:

This `Clock`'s time has been moved forward by the indicated number of minutes. Note: A negative argument moves this `Clock` backward.

◆ **getHour**

`public int getHour( )`

Get the current hour of this `Clock`.

Returns:

the current hour (always in the range 1...12)

◆ **getMinute**

`public int getMinute( )`

Get the current minute of this `Clock`.

Returns:

the current minute (always in the range 0...59)

◆ **isMorning**

`public boolean isMorning( )`

Check whether this `Clock`'s time is before noon.

Returns:

If this `Clock`'s time lies from 12:00 midnight to 11:59 A.M. (inclusive), then the return value is `true`; otherwise the return value is `false`.

◆ **setTime**

`public void setTime(int hour, int minute, boolean morning)`

Set the current time of this `Clock`.

Parameters:

`hour` – the hour at which to set this `Clock`
`minute` – the minute at which to set this `Clock`
`morning` – indication of whether the new time is before noon

Precondition:

1 <= `hour` <= 12, and 0 <= `minute` <= 59.

Postcondition:

This `Clock`'s time has been set to the given hour and minute (using the usual 12-hour time notation). If the third parameter, `morning`, is `true`, then this time is from 12:00 midnight to 11:59 A.M. Otherwise this time is from 12:00 noon to 11:59 P.M.

Throws: `IllegalArgumentException`

Indicates that the `hour` or `minute` is illegal.

The solution to the clock problem is a new concept, called **extended classes**, described here:

Extended Classes

Extended classes use a concept called **inheritance**. In particular, once we have a class, we can then declare new classes that contain all of the methods and instance variables of the original class—plus any extras that you want to throw in. This new class is called an extended **class** of the original class. The original class is called the **superclass**. And the methods that the extended class receives from its superclass are called **inherited** methods.

How to Declare an Extended Class

In the declaration of an extended class, the name of the extended class is followed by the keyword `extends`, and then the name of the superclass. For example, suppose that we want to declare an extended class `CuckooClock` using the existing `Clock` class as the superclass. The beginning of the `CuckooClock` class declaration would then look like this:

```
public class CuckooClock extends Clock
{
    ...
```

This declaration indicates that every `CuckooClock` is also an ordinary `Clock`. The primary consequence is that all of the public methods and instance variables of an ordinary `Clock` are immediately available as public members of a `CuckooClock`. These members are said to be **inherited** from the `Clock`. Notice that it is the *public* members of the `Clock` that are accessible to the `CuckooClock`. In some sense, the private members of the `Clock` are also present in a `CuckooClock`—they must be present because some of the public methods access other private members. But, these private members cannot be accessed directly in a `CuckooClock`, not even by the programmer who implements a `CuckooClock`.

For future reference, you should know that there is another kind of access that can be provided to a method or instance variable. The access is called **protected**. In most respects, a protected member is just like a private member, but the programmer of an extended class has direct access to protected members. None of our examples will use protected members.

Now, let's finish our `CuckooClock` declaration and see how it can be used in a program. Our complete `CuckooClock` is shown in Figure 13.2. As you can see, a `CuckooClock` has one extra public method: a boolean method called `isCuckooing`, which returns `true` when the clock's cuckoo is making noise. In the implementation of `isCuckooing`, our cuckoos make noise whenever the current minute of the time is zero. In this implementation, we use the ordinary clock method, `getMinute`, to determine whether the current minute is zero.

Once the `CuckooClock` declaration is available, a program may declare `CuckooClock` objects using all the public `Clock` methods and also using the new `isCuckooing` method. In this usage, there are some special considerations for the constructors. We'll discuss these considerations next.

FIGURE 13.2 The `CuckooClock` Is an Extension of the Ordinary `Clock`

Class CuckooClock

❖ **public class CuckooClock from the package edu.colorado.simulations**
➤ **extends Clock**

A `CuckooClock` is a `Clock` that cuckoos when the minute is zero. The primary purpose of this class is to demonstrate how an extended class is implemented.

Specification

In addition to the `Clock` methods, a `CuckooClock` has:

- **isCuckooing**

`public boolean isCuckooing( )`

Check whether this `CuckooClock` is currently cuckooing.

Returns:

If this `CuckooClock`'s current minute is zero, then the return value is `true`; otherwise the return value is `false`.

Implementation

```
// File: CuckooClock.java from the package edu.colorado.simulations
// Documentation is available at the top of this diagram or from the CuckooClock link in
//    http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;

public class CuckooClock extends Clock
{

   public boolean isCuckooing( )
   {
      return (getMinute( ) == 0);
   }

}
```

The Constructors of an Extended Class

An extended class may declare its own constructors, or it may use a no-arguments constructor that is inherited from the superclass. Other superclass constructors, with arguments, are not inherited by the extended class. For the cuckoo clock, the inherited no-arguments constructor, from the ordinary clock, is sufficient. For example, a program can allocate a new `CuckooClock` object with the statement:

```
CuckooClock noisy = new CuckooClock( );
```

This activates the inherited no-arguments constructor from the `Clock` to create a new `CuckooClock`.

Several important aspects of constructors for an extended class are:

1. If an extended class has new instance variables that are not part of the superclass, then an inherited no-arguments constructor will set these new values to their default values (such as zero for an `int`), and then do the work of the superclass's constructor.
2. Other superclass constructors, with arguments, are not inherited by the extended class.
3. If an extended class declares any constructors of its own, then none of the superclass constructors are inherited, not even the no-arguments constructor. Later we will examine the special format that should be used to write an extended class with its own declared constructors.

Using an Extended Class

We now know enough to write a bit of code that uses an extended class. For example, here is a bit of code that declares a cuckoo clock, advances the cuckoo clock some number of minutes, and then prints a message about whether the cuckoo clock is currently cuckooing:

```
CuckooClock noisy = new CuckooClock( );

noisy.advance(42);

System.out.print("The noisy clock’s time is now ");
System.out.println
   (noisy.getHour( ) + ":" + noisy.getMinute( ));

if (noisy.isCuckooing( ))
   System.out.println("Cuckoo cuckoo cuckoo.");
else
   System.out.println("All’s quiet on the cuckoo front.");
```

The key feature is that a cuckoo clock may use ordinary clock methods (such as `advance`), and it may also use the new `isCuckooing` method. The inheritance is accomplished with little work on our part. We only need to write the body of `isCuckooing`; none of the ordinary clock methods needs to be rewritten for the cuckoo clock.

There's another advantage to extended classes: An object of the extended class may be used at any location where the superclass is expected. For example, suppose you write a method to compare the times on two clocks, as shown in Figure 13.3. This method could be part of the `Clock` class or it could be

FIGURE 13.3 A Method to Compare the Time of Two Clocks

Implementation

```
public static boolean earlier (Clock c1, Clock c2)
// Postcondition: Returns true if the time on c1 is earlier than the time on c2
// over a usual day (starting at midnight and continuing to 11:59 P.M.); otherwise returns false.
{
   // Check whether one is morning and the other is not.
   if (c1.isMorning( ) && !c2.isMorning( ))
      return true;
   else if (c2.isMorning( ) && !c1.isMorning( ))
      return false;

   // Check whether one is 12 o'clock and the other is not.
   else if ((c1.getHour( ) == 12) && (c2.getHour( ) != 12))
      return true;
   else if ((c2.getHour( ) == 12) && (c1.getHour( ) != 12))
      return false;

   // Check whether the hours are different from each other.
   else if (c1.getHour( ) < c2.getHour( ))
      return true;
   else if (c2.getHour( ) < c1.getHour( ))
      return false;

   // The hours are the same, so check the minutes.
   else if (c1.getMinute( ) < c2.getMinute( ))
      return true;
   else
      return false;
}
```

implemented elsewhere. We can use this method to compare two ordinary clocks, but we can also compare objects of the clock's extended classes. For example, consider the code shown here:

```
CuckooClock sunrise = new CuckooClock( );
CuckooClock yourTime = new CuckooClock( );

...code that sets the clocks to some time...

if (earlier(sunrise, yourTime))
   System.out.println("That's before sunrise!");
else
   System.out.println("That's not before sunrise.");
```

In fact, we can even use the `earlier` method to compare an ordinary `Clock` with a `CuckooClock`, or to compare two objects from different extended classes.

Any methods that you write to manipulate a clock will also be able to manipulate all of the clock's extended classes. Without extended classes, we would need to write a separate method for each kind of clock that we want to manipulate.

Overriding Inherited Methods

Frequently, an extended class needs to perform some method differently from the way the superclass does. For example, the original clock provides the current hour via `getHour`, using a 12-hour clock. Suppose we want to implement an extended class that provides its hour on a 24-hour basis, ranging from 0 to 23. The new clock can be defined as an extended class called `Clock24`. The `Clock24` class inherits everything from the ordinary clock, but it provides a new `getHour` method. This is called **overriding** an inherited method.

To override an inherited method, the extended class redefines the method within its own class declaration. For example, the `Clock24` class in Figure 13.4 overrides the original `getHour` method of the original `Clock`. Within this implementation, we can use the name `super.getHour` to refer to the original `getHour` method of the superclass.

TIP

Programming Tip: Make the Overriding Method Activate the Original

In the example of the `Clock24 getHour` method, the first action of the overriding method is to activate the original method. Generally, the overriding method will activate the original method to do some of its work. The reason is that the original method does work to correctly maintain the superclass. Frequently the activation of the original method will be the *first* action of an overriding method.

FIGURE 13.4 The Clock24 Class Overrides the Clock's `getHour` Method

Class Clock24

❖ **public class Clock24 from the package edu.colorado.simulations**
➤ **extends Clock**

A `Clock24` object is a `Clock` with its hour in 24-hour format (0 to 23) instead of 12-hour format. The purpose is to show how an extended class may override a method of the superclass.

Specification

In addition to the `Clock` methods, a `Clock24` object has:

- **getHour (overriden from the superclass Clock)**

  `public int getHour( )`

  Get the current hour of this `Clock`24, in 24-hour format.

  Returns:
  the current hour (always in the range 0...23)

Implementation

```
// File: Clock24.java from the package edu.colorado.simulations
// Documentation is available at the top of this diagram or from the Clock24 link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;

public class Clock24 extends Clock
{
   public int getHour( )
   {
      int ordinaryHour = super.getHour( );

      if (isMorning( ))
      {
         if (ordinaryHour == 12)
            return 0;
         else
            return ordinaryHour;
      }
      else
      {
         if (ordinaryHour == 12)
            return 12;
         else
            return ordinaryHour + 12;
      }
   }

}
```

Widening Conversions for Extended Classes

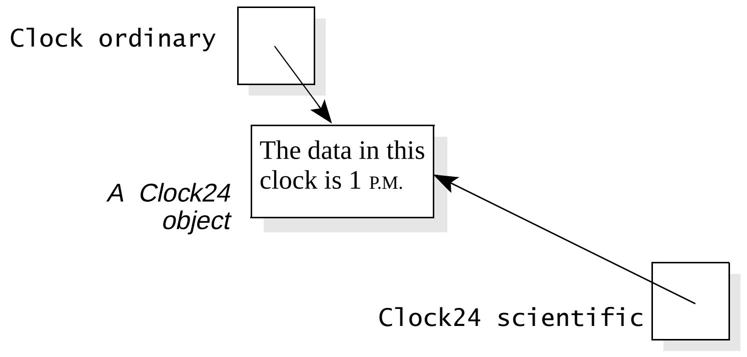
In Java, assignments are allowed from an extended class to the superclass, for example:

```
Clock ordinary;
Clock24 scientific = new Clock24( );

scientific.advance(780);
ordinary = scientific;
```

Advance the scientific clock to 1 P.M. and assign the ordinary clock to equal the scientific clock.

The assignment `ordinary = scientific` is an example of a *widening conversion*, which we first saw in Chapter 5 on page 243. It is a widening conversion because the variable `ordinary` is capable of referring to many different kinds of clocks (an ordinary `Clock`, or a `CuckooClock`, or a `Clock24`), thus the `ordinary` variable has a wider variety of possibilities than the `scientific` variable (which may refer to a `Clock24` object only). Assigning an object of the extended class to a variable of the superclass is always permitted, and the assignment acts like any other Java assignment. After the above statements, both `ordinary` and `scientific` refer to the same clock, and this clock is the `Clock24` object that was allocated and advanced 780 minutes.

During the execution of a Java program, the Java runtime system keeps track of the class of each object. In the above code, the Java runtime system knows that the one clock object is really a `Clock24` object, so we might diagram the above situation like this (after the assignment statement):



What happens if we activate `scientific.getHour( )`? The `Clock24` `getHour` method is activated (from Figure 13.4 on page 625), and the answer 13 is returned (which is the hour corresponding to 1 P.M.). That's not surprising since `scientific` was declared as a `Clock24` variable.

But what happens when we activate `ordinary.getHour( )`? In this situation, the answer is still 13, even though `ordinary` is declared as a `Clock` variable rather than a `Clock24` variable. Here's the reason:

How Java Methods Are Activated

When a program is running and a method is activated, the Java Runtime System checks the data type of the actual object, and uses the method from that type (rather than the method from the type of the reference variable).

This technique of method activation is called **dynamic method activation** because the actual method to activate is not determined until the program is running. A method that behaves like this is called a **virtual method**. Some programming languages, such as C++, allow both virtual methods (where the method to activate is determined by the actual object when the program is running) and nonvirtual methods (where the method to activate is determined by the type of the variable during the compilation). But Java has virtual methods only.

Narrowing Conversions for Extended Classes

Assignments are also permitted from a superclass to one of its extended types, but the compiler needs a bit of extra reassurance to permit the assignment. For example, suppose that `c` is a `Clock` variable and `fancy` is a `CuckooClock` variable. To make an assignment from `c` to `fancy` we must write

```
fancy = (CuckooClock) c;
```

The typecast `(CuckooClock) c` tells the compiler that the reference variable `c` is really referring to a `CuckooClock` object, even though it was declared as a `Clock`. With the typecast in place, the assignment always compiles correctly, though during the execution there is an extra check. When the assignment is executed, the Java Runtime System checks that `c` really does refer to a `CuckooClock`, and if it doesn't then a `ClassCastException` is thrown.

The assignment `fancy = (CuckooClock) c` is an example of a *narrowing conversion*, which we first saw in Chapter 5 on page 244. In Java, a narrowing conversion always requires a typecast to reassure the compiler that you really did mean to make that assignment. Even with the typecast, a programming error can cause a `ClassCastException` at runtime if the object is the wrong type.

Several features of extended classes remain to be seen, such as extended classes that require new private instance variables. These considerations will arise in the examples from the rest of this chapter.

Self-Test Exercises

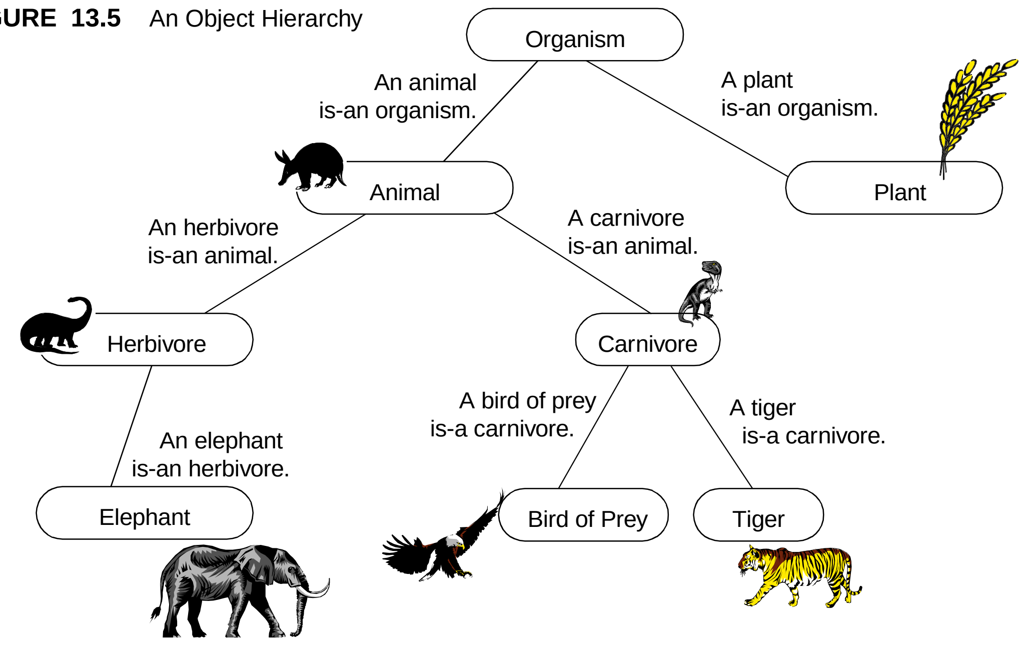
1. Design and implement an extended class called `DaylightClock`. A daylight clock is like a clock except that it has one extra boolean method to determine whether it is currently daylight. Assume that daylight stretches from 7 A.M. through 6:59 P.M.
2. Using your `DayLightClock` from the previous exercise, write an example of a widening conversion and a narrowing conversion.
3. Suppose an extended class does not declare any constructors of its own. What constructors is it given automatically?
4. Design and implement an extended class called `NoonAlarm`. A noon alarm object is just like a clock, except that whenever the `advance` method is called to advance the clock forward through 12 o'clock noon, an alarm message is printed (by the `advance` method).

13.2 SIMULATION OF AN ECOSYSTEM

A is-a B means that A is a particular kind of B

There are many potential uses for extended classes, but one of the most frequent uses comes from the is-a relationship. "A is-a B" means that each A object is a particular kind of B object. For example, a cuckoo clock is a particular kind of clock. Some other examples of is-a relationships for living organisms are shown in Figure 13.5. The relationships are drawn in a tree called an **object hierarchy** tree. In this tree, each superclass is placed as the parent of its extended classes.

FIGURE 13.5 An Object Hierarchy


Organism
An animal is-an organism.
A plant is-an organism.
Animal
Plant
An herbivore is-an animal.
A carnivore is-an animal.
Herbivore
Carnivore
An elephant is-an herbivore.
A bird of prey is-a carnivore.
A tiger is-a carnivore.
Elephant
Bird of Prey
Tiger

Programming Tip: **When to Use an Extended Class**

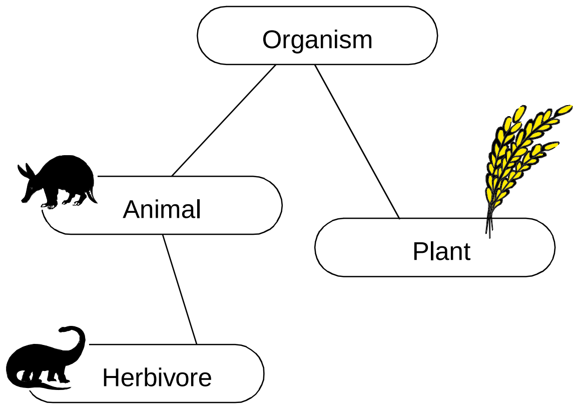
TIP

Look at each line in the object hierarchy tree (Figure 13.5). For each child and its parent, does it make sense to say "A is a B"? Whenever it makes sense to say "A is a B," consider implementing the class for A as an extended class of B. This lets the new A class benefit from inheriting all of B's public members.

Implementing Part of the Organism Object Hierarchy

We will implement four classes from the object hierarchy tree of living organisms, and use these four classes in a program that simulates a small ecosystem. The four classes that we will implement are:

- A general class, called `Organism`, that can be used by a program to simulate the simplest properties of organisms, such as being born, growing, and eventually dying.
- Two classes that are extended from an `Organism`. The classes, called `Animal` and `Plant`, can do everything that an ordinary organism can do—but they also have extra abilities associated with animals and plants.
- The final class, called `Herbivore`, is extended from the animal class. It is a special kind of animal that eats plants.



Keep in mind that the extended classes might have new instance variables as well as new methods.

The Organism Class

At the top of our object hierarchy tree is a class called `Organism`. Within a program, every organism is given an initial size, measured in ounces. Each organism is also given a growth rate, measured in ounces per week. A program that wants to simulate the growth of an organism will start by specifying an initial size and growth rate as arguments to the organism's constructor. Throughout the computation, the program may activate a method called `simulateWeek`, which causes the organism to simulate the passage of one week in its life—in other words, activating `simulateWeek` makes the organism grow by its current growth rate. The `Organism` class has a few other methods specified in Figure 13.6 on page 630, and a usage of the `Organism` class is shown in Figure 13.7 on page 632.

FIGURE 13.6 Specification for the `Organism` Class

Class Organism

❖ **public class Organism from the package edu.colorado.simulations**

An `Organism` object simulates a growing organism such as a plant or animal.

Specification

◆ **Constructor for the Organism**

`public Organism(double initSize, double initRate)`

Construct an `Organism` with a specified size and growth rate.

Parameters:

`initSize` – the initial size of this `Organism`, in ounces
`initRate` – the initial growth rate of this `Organism`, in ounces per week

Precondition:

`initSize >= 0`. Also, if `initSize` is zero, then `initRate` must also be zero.

Postcondition:

This `Organism` has been initialized. The value returned from `getSize( )` is now `initSize`, and the value returned from `getRate( )` is now `initRate`.

Throws: `IllegalArgumentException`

Indicates that `initSize` or `initRate` violates the precondition.

◆ **alterSize**

`public void alterSize(double amount)`

Change the current size of this `Organism` by a given amount.

Parameters:

`amount` – the amount to increase or decrease the size of this `Organism` (in ounces)

Postcondition:

The given amount (in ounces) has been added to the size of this `Organism`. If this new size is less than or equal to zero, then `expire` is also activated.

◆ **expire**

`public void expire( )`

Set this `Organism`'s size and growth rate to zero.

Postcondition:

The size and growth rate of this `Organism` have been set to zero.

(continued)

(FIGURE 13.6 continued)

◆ **getRate**

`public double getRate( )`

Get the growth rate of this `Organism`.

Returns:
the growth rate of this `Organism` (in ounces per week)

◆ **getSize**

`public double getSize( )`

Get the current size of this `Organism`.

Returns:
the current size of this `Organism` (in ounces)

◆ **isAlive**

`public boolean isAlive( )`

Determine whether this `Organism` is currently alive.

Returns:
If this `Organism`'s current current size is greater than zero, then the return value is `true`; otherwise the return value is `false`.

◆ **setRate**

`public void setRate(double newRate)`

Set the current growth rate of this `Organism`.

Parameters:
`newRate` – the new growth rate for this `Organism` (in ounces per week)

Precondition:
If the size is currently zero, then `newRate` must also be zero.

Postcondition:
The growth rate for this `Organism` has been set to `newRate`.

Throws: `IllegalArgumentException`
Indicates that the size is currently zero, but the `newRate` is nonzero.

◆ **simulateWeek**

`public void simulateWeek( )`

Simulate the passage of one week in the life of this `Organism`.

Postcondition:
The size of this `Organism` has been changed by its current growth rate. If the new size is less than or equal to zero, then `expire` is activated to set both size and growth rate to zero.

FIGURE 13.7 Sample Program from the Movie, *The Blob*

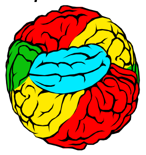
Java Application Program

```
// FILE: Blob.java
// This small demonstration shows how the Organism class is used.
import edu.colorado.simulations.Organism

public class Blob
{
   public static void main(String[ ] args)
   {
      Organism blob = new Organism(20.0, 100000.0);
      int week;

      // Untroubled by conscience or intellect, the Blob grows for three weeks.
      for (week = 1; week <= 3; week++)
      {
         blob.simulateWeek( );
         System.out.print("Week " + week + ":" + " the Blob is ");
         System.out.println(blob.getSize( ) + " ounces.");
      }

      // Steve McQueen reverses the growth rate to -80000 ounces per week.
      blob.setRate(-80000.0);
      while (blob.isAlive( ))
      {
         blob.simulateWeek( );
         System.out.print("Week " + week + ":" + " The Blob is ");
         System.out.println(blob.getSize( ) + " ounces.");
         week++;
      }
      System.out.println("The Blob (or its son) shall return.");
   }

}
```

Sample Dialogue

```
Week 1: The Blob is 100020.0 ounces.
Week 2: The Blob is 200020.0 ounces.
Week 3: The Blob is 300020.0 ounces.
Week 4: The Blob is 220020.0 ounces.
Week 5: The Blob is 140020.0 ounces.
Week 6: The Blob is 60020.0 ounces.
Week 7: The Blob is 0.0 ounces.
The Blob (or its son) shall return.
```

Can anyone stop the Blob?!



Steve McQueen comes to the rescue at the end of week 3!

The `Organism` class has methods to set a new growth rate, a method to alter the organism's current size, and methods to return information about the organism's current size and growth rate.

The organism class is not hard to implement, and we'll leave its implementation up to you. But we will give one example of using the organism class. Movie buffs may recall the 1958 film, *The Blob*. The Blob came to Earth from outer space at a mere 20 ounces, but "untroubled by conscience or intellect," it absorbs anything and anyone in its path. Without giving away the whole plot, let's suppose that the Blob grows at the astonishing rate of 100,000 ounces per week for three weeks. Then our hero (Steve McQueen) manages to reverse its growth to a rate of negative 80,000 ounces per week. A program to simulate the movie plot is shown in Figure 13.7.

The Animal Class: An Extended Class with New Private Instance Variables

Now we want to implement a class that can be used to simulate an animal. Since an animal *is-an* organism, it makes sense to declare the `Animal` class as an extended class of the `Organism`. In our design, an animal is an organism that must consume a given amount of food each week to survive. If a week has passed and the animal has consumed less than its required amount of food, then death occurs. With this in mind, the `Animal` class will have two new private instance variables, which are not part of the `Organism` class, as shown at the front of this partial declaration:

```
public class Animal extends Organism
{
   private double needEachWeek;
   private double eatenThisWeek;

   || We discuss the animal's public methods in a moment.
   ...
```

the extended class has two new private instance variables

The first new instance variable, `needEachWeek`, keeps track of how many ounces of food the animal must eat each week in order to survive. The second new instance variable, `eatenThisWeek`, keeps track of how many ounces of food the animal has eaten so far this week.

When an extended class has some new instance variables, it will usually need a new constructor to initialize those instance variables. This is the first example that we have seen where an extended class has a new constructor rather than using the inherited constructors that were described on page 622.

How to Provide a New Constructor for an Extended Class

When an extended class has a new constructor, the implementation of the new constructor appears in the class declaration, just like any other constructor. The inherited no-arguments constructor can no longer be used to create an object of the extended class.

In the case of the animal, the new constructor will have three arguments. The first two arguments are the same as the arguments for any organism, providing the initial size and the initial growth rate. The third argument will indicate how much food the animal needs, in ounces per week. Thus, the start of the animal's declaration is given here:

```
public class Animal extends Organism
{
   public double needEachWeek;
   public double eatenThisWeek;

   public Animal
   (double initSize, double initRate, double initNeed)
   ...
```

The constructor has no argument for `eatenThisWeek`, since we plan to have that instance variable initialized to zero, indicating that a newly constructed animal has not yet eaten.

The work of the animal's constructor is easy enough to describe: The first two arguments must somehow initialize the size and growth rate of the animal; the last argument initializes `needEachWeek`; the value of `eatenThisWeek` is initialized to zero. But how do we manage to use `initSize` and `initRate` to initialize the size and growth rate of the animal? Most likely the size and growth rate are stored as private instance variables of the `Organism` class, but the animal has no direct access to the organism's private instance variables.

Java provides a solution to this problem, called a **super constructor.** A super constructor is any constructor of the superclass. Usually, a constructor for an extended class will activate the super constructor in its first line of code. This is done with the keyword `super`, followed by the argument list for the super constructor.

Here is the implementation of our animal constructor, with the activation of the super constructor highlighted:

```
public Animal
(double initSize, double initRate, double initNeed)
{
   super(initSize, initRate);
   if (initNeed < 0)
      throw new IllegalArgumentException("negative need");
   needEachWeek = initNeed;
   // eatenThisWeek will be given its default value of zero.
}
```

If a constructor for an extended class does not activate the super constructor, then Java arranges for the superclass's no-arguments constructor to be automatically activated at the start of the extended class's constructor.

The Other Animal Methods

The animal has four new methods that deal with eating, and the `simulateWeek` method must also be overridden. The four new methods are called `assignNeed`, `eat`, `stillNeed`, and `getNeed`. We'll discuss each of these methods now.

The `assignNeed` method. This method has this heading:

```
void assignNeed(double newNeed)
```

The method is activated when a simulation needs to specify how much food an animal must eat to survive a week. For example, if `spot` is an animal that needs 30 ounces of food to survive a week, then `spot.assignNeed(30)` is activated. During a simulation, the food requirements may change, so that `assignNeed` can be activated several times with different arguments.

The `eat` method. Whenever the animal, `spot`, eats `m` ounces of food, the method `spot.eat(m)` records this event. Here's the heading of the method:

```
void eat(double amount)
```

The amount of food that has been eaten during the current week is stored in a private instance variable, `eatenThisWeek`. So, activating `eat(m)` will simply add `m` to `eatenThisWeek`.

Two accessor methods. There are two animal methods that are accessor methods called `getNeed` and `stillNeed`. The `getNeed` method returns the total amount of food that the animal needs in a week, and the `stillNeed` method returns the amount of food that the animal still needs in the current week (which is the total need minus the amount already eaten).

Overriding the `simulateWeek` method. The animal must do some extra work in its `simulateWeek` method. Therefore, it will override the organism's `simulateWeek` method. The animal's `simulateWeek` will first activate `super.simulateWeek` to carry out whatever work an ordinary organism does to simulate one week. Next, the animal's `simulateWeek` determines whether the animal has had enough food to eat this week. If `eatenThisWeek` is less than `needEachWeek`, then `expire` is activated. Also, `eatenThisWeek` is reset to zero to restart the recording of food eaten for the animal's next week.

At the top of the next page, we show some example code to illustrate the coordination of the new methods. It begins by declaring a 160-ounce animal, `spot` (perhaps a cat). Spot is not currently growing (since `initRate` is zero in the constructor), but she does require 30 ounces of food per week.

```
Animal spot = new Animal(160, 0, 30);

spot.eat(10);
spot.eat(25);
spot.simulateWeek( );
if (spot.isAlive( ))
   System.out.println("Spot lives!");
else
   System.out.println("Spot has died.");

spot.eat(10);
spot.eat(15);
spot.simulateWeek( );
if (spot.isAlive( ))
   System.out.println("Spot lives!");
else
   System.out.println("Spot has died.");
```

Spot catches a 10-ounce fish and steals 25 ounces of chicken from the kitchen.

Spot still lives at the end of her first week.

Spot catches another 10-ounce fish, but gets only 15 ounces of chicken this week.

Sadly, Spot dies at the end of her second week.

The specification and implementation for the animal appears in Figure 13.8. Since the animal is in the same package as the organism (`edu.colorado.simulations`), there is no need to import the `Organism` class.

The next class that we'll build is a class to simulate a plant. The `Plant` class is extended from an organism, and it has one extra method—but the work is left up to you in the next few exercises.

Self-Test Exercises

5. Draw an object hierarchy diagram for various kinds of people.
6. Declare a new class called `Plant`, extended from `Organism` with one extra method:

   ```
   void NibbledOn(double amount)
   // Precondition: 0 <= amount <= getSize( ).
   // Postcondition: The size of this Plant has been decreased by
   // amount. If this reduces the size to zero, then expire is activated.
   ```

   Suppose `fern` is a plant. Activating `fern.NibbledOn(m)` corresponds to some beast eating `m` ounces of `fern`. Notice that `NibbledOn` differs from the existing `alterSize` method, since in the `NibbledOn` method, the amount is removed from the size (rather than adding to the size), and there is also a strict precondition on the amount eaten. The `nibbledOn` implementation should activate `alterSize` to do some of its work.

   Your `Plant` class should have one constructor with the same parameters as the `Organism` constructor. The plant's constructor merely activates the super constructor with these same parameters.

7. Write a static method with one argument, which is a Java `Vector`, as shown in this header:

   ```
   public static double totalMass(Vector organisms)
   ```

   The method's precondition requires that every object in the `Vector` is a non-null `Organism`. The return value is the total mass of all these organisms. You may need to read more about the `Vector` class in Appendix D.

FIGURE 13.8 The `Animal` Class

Class Animal

❖ **public class Animal from the package edu.colorado.simulations**
➤ **extends Organism**

An `Animal` is an `Organism` with extra methods that deal with eating.

Specification

In addition to the `Organism` methods, an `Animal` has a new constructor and these new methods:

◆ **Constructor for the Animal**

`public Animal(double initSize, double initRate, double initNeed)`

Construct an `Animal` with a specified size, growth rate, and weekly eating need.

Parameters:
`initSize` – the initial size for this `Animal`, in ounces
`initRate` – the initial growth rate for this `Organism`, in ounces per week
`initNeed` – the initial weekly eating requirement for this `Animal`, in ounces per week

Precondition:
`initSize >= 0` and `initNeed >= 0`. Also, if `initSize` is zero, then `initRate` must also be zero.

Postcondition:
This `Animal` has been initialized. The value returned from `getSize( )` is now `initSize`, the value returned from `getRate( )` is now `initRate`, and this `Animal` must eat at least `initNeed` ounces of food each week to survive.

Throws: `IllegalArgumentException`
Indicates that `initSize`, `initRate`, or `initNeed` violates the precondition.

(continued)

(FIGURE 13.8 continued)

- **eat**

  `public void eat(double amount)`

  Have this `Animal` eat a given amount of food.

  Parameters:
  `amount` – the amount of food for this `Animal` to eat (in ounces)

  Precondition:
  `amount` >= 0.

  Postcondition:
  The amount (in ounces) has been added to the food that this `Animal` has eaten this week.

  Throws: `IllegalArgumentException`
  Indicates that `amount` is negative.

- **getNeed**

  `public double getNeed( )`

  Determine the amount of food that this `Animal` needs each week.

  Returns:
  the total amount of food that this `Animal` needs to survive one week (measured in ounces)

- **setNeed**

  `public void setNeed(double newNeed)`

  Set the current growth weekly food requirement of this `Animal`.

  Parameters:
  `newNeed` – the new weekly food requirement for this `Animal` (in ounces)

  Precondition:
  `newNeed >= 0`.

  Postcondition:
  The weekly food requirement for this `Animal` has been set to `newNeed`.

  Throws: `IllegalArgumentException`
  Indicates that `newNeed` is negative.

- **simulateWeek (overriden from the superclass Organism)**

  `public void simulateWeek( )`

  Simulate the passage of one week in the life of this `Animal`.

  Postcondition:
  The size of this `Animal` has been changed by its current growth rate. If the new size is less than or equal to zero, then `expire` is activated to set both size and growth rate to zero. Also, if this `Animal` has eaten less than its need over the past week, then `expire` has been activated.

- **stillNeed**

  `public double stillNeed( )`

  Determine the amount of food that this `Animal` still needs to survive this week.

  Returns:
  the ounces of food that this `Animal` still needs to survive this week
  (which is the total need minus the amount eaten so far)

(continued)

(FIGURE 13.8 continued)

Implementation

```
// File: Animal.java from the package edu.colorado.simulations
// Documentation is available on pages 637-638 or from the Animal link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;

public class Animal extends Organism
{
   private double needEachWeek;   // Amount of food needed (in ounces per week)
   private double eatenThisWeek;  // Ounces of food eaten so far this week

   public Animal(double initSize, double initRate, double initNeed)
   {
      super(initSize, initRate);
      if (initNeed < 0)
         throw new IllegalArgumentException("negative need");
      needEachWeek = initNeed;
      // eatenThisWeek will be given its default value of zero.
   }

   public void eat(double amount)
   {
       if (amount < 0)
          throw new IllegalArgumentException("amount is negative");
       eatenThisWeek += amount;
   }

   public double getNeed( )
   {
      return needEachWeek;
   }

   public void setNeed(double newNeed)
   {
       if (newNeed < 0)
          throw new IllegalArgumentException("newNeed is negative");
       needEachWeek = newNeed;
   }
```

(continued)

(FIGURE 13.8 continued)

```
      public void simulateWeek( )
      {
         super.simulateWeek( );
         if (eatenThisWeek < needEachWeek)
            expire( );
         eatenThisWeek = 0;
      }

      public double stillNeed( )
      {
         if (eatenThisWeek >= needEachWeek)
            return 0;
         else
            return needEachWeek - eatenThisWeek;
      }
   }
```

The Herbivore Class

We're almost ready to start designing a simulation program for a small ecosystem. The ecosystem will be a small pond containing weeds and weed-eating fish. The weeds will be modeled by the `Plant` class from Self-Test Exercise 6 on page 636, and the fish will be a new class that is extended from the animal class that we have just completed.

The new class for the fish, called `Herbivore`, is an animal that eats plants. This suggests that an herbivore should have one new method, which we call `nibble`. The method will interact with a plant that the herbivore is nibbling, and this plant is a parameter to the new method. Here is the specification:

◆ **nibble (from the Herbivore class)**

`public void nibble(Plant meal)`

Have this `Herbivore` eat part of a `Plant`.

Parameters:

`meal` - the `Plant` that will be partly eaten

Postcondition:

Part of the `Plant` has been eaten by this `Herbivore`, by activating both `eat(amount)` and `meal.nibbledOn(amount)`. The amount is usually half of the `Plant`, but it will not be more than 10% of this `Herbivore`'s weekly need nor more than the amount that this `Herbivore` still needs to eat to survive this week.

For example, suppose that `carp` is an `Herbivore`, and `bushroot` is a `Plant`. If we activate `carp.nibble(bushroot)`, then `carp` will eat some of `bushroot`, by activating two other methods: (1) its own `eat` method, and (2) the `bushroot.nibbledOn( )` method.

The `nibble` method follows a few rules about how much of the plant is eaten. The rules state that `carp.nibble(bushroot)` will usually cause `carp` to eat half of `bushroot`, but a single nibble will not eat more than 10% of the herbivore's weekly need nor more than the amount that the herbivore still needs to eat in order to survive the rest of the week. In an actual model, these rules would be determined from behavior studies of real herbivores.

The complete herbivore documentation is shown in Figure 13.9, along with the implementation of the herbivore's methods.

FIGURE 13.9 The Herbivore Class

Class Herbivore

❖ **public class Herbivore from the package edu.colorado.simulations**
➤ **extends Animal**

An `Herbivore` is an `Animal` with an extra method for eating a `Plant`.

Specification

In addition to the `Animal` methods, an `Organism` has a new constructor and the following new methods:

◆ **Constructor for the Herbivore**

`public Herbivore(double initSize, double initRate, double initNeed)`

This is the same as the `Animal` constructor.

◆ **nibble (from the Herbivore class)**

`public void nibble(Plant meal)`

Have this `Herbivore` eat part of a `Plant`.

Parameters:
`meal` - the `Plant` that will be partly eaten

Postcondition:
Part of the `Plant` has been eaten by this `Herbivore`, by activating both `eat(amount)` and `meal.nibbledOn(amount)`. The amount is usually half of the `Plant`, but it will not be more than 10% of this `Herbivore`'s weekly need nor more than the amount that this `Herbivore` still needs to eat to survive this week.

(continued)

(FIGURE 13.9 continued)

Implementation

```
// File: Herbivore.java from the package edu.colorado.simulations
// Documentation is available on the preceding page or from the Herbivore link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;

public class Herbivore extends Animal
{
   public Herbivore(double initSize, double initRate, double initNeed)
   {
      super(initSize, initRate, initNeed);
   }

   public void nibble(Plant meal)
   {
      final double PORTION = 0.5;       // Eat no more than this portion of
                                        // the plant
      final double MAX_FRACTION = 0.1; // Eat no more than this fraction of
                                        // the weekly need

      double amount;                    // How many ounces of the plant will
                                        // be eaten

      // Set amount to some portion of the Plant, but no more than a given maximim fraction
      // of the total weekly need, and no more than what this Herbivore still needs to eat this
      // week.
      amount = PORTION * meal.getSize( );
      if (amount > MAX_FRACTION * getNeed( ))
         amount = MAX_FRACTION * getNeed( );
      if (amount > stillNeed( ))
         amount = stillNeed( );

      // Eat the Plant.
      eat(amount);
      meal.nibbledOn(amount);
   }
}
```

The Pond Life Simulation Program

A simulation program can use objects such as our herbivores to predict the effects of changes to an ecosystem. We'll write a program along these lines to model the weeds and fish in a small pond. The program stores the pond weeds in a collection of plants. To be more precise, we have Java `Vector` that will contain all the `Plant` objects of the simulation program. Check Appendix D if you need information on the `Vector` collection class.

For example, suppose the pond has 2000 weeds with an initial size of 15 ounces each and a growth rate of 2.5 ounces per week. Then we will create a `Vector` of 2000 plants, as shown here:

```
public static final int    MANY_WEEDS = 2000;
public static final double WEED_SIZE  =   15;
public static final double WEED_RATE  =  2.5;
...
Vector weeds = new Vector(MANY_WEEDS);

int i; // Loop control variable

for (i = 0; i < MANY_WEEDS; i++)
   weeds.addElement(new Plant(WEED_SIZE, WEED_RATE));
```

Let's start our explanation of this code with the highlighted expression `new Plant(WEED_SIZE, WEED_RATE)`. This expression uses the `new` operator to allocate a new `Plant` object. After the type name, `Plant`, we have an argument list `(WEED_SIZE, WEED_RATE)`. This is the form of `new` that allocates a new plant using a constructor with two arguments `WEED_SIZE` and `WEED_RATE`. The statement `weeds.addElement(new Plant(WEED_SIZE, WEED_RATE))` is executed 2000 times in the code. Each of the 2000 allocations results in a new plant, and references to these 2000 plants are placed in the `weeds Vector`.

Our simulation has a second `Vector`, called `fish`, which is a `Vector` of `Herbivore` objects. Initially, we'll stock the fish `Vector` with 300 full-grown fish.

With the weeds and fish in place, our simulation may proceed. Throughout the simulation, various fish nibble on various weeds. Each week, every weed increases by its growth rate (stated as 2.5 ounces/week in the code). Some weeds will also be nibbled by fish, but during our simulation no weed will ever be completely eaten, so the weeds never die, nor do we ever create new weeds beyond the initial 2000. On the other hand, the number of fish in the pond may vary throughout the simulation. When a fish dies (because of insufficient nibbling), the reference to that fish is removed from the fish `Vector`. New fish are also born each week at a rate that we'll explain in a moment. For now, though, you should be getting a good idea of the overall simulation. Let's lay out these ideas precisely with some pseudocode.

```
// Pseudocode for the pond life simulation
  1. Create a bunch of new plants, and put the references to these new plants in
     a Vector called weeds. The exact number of weeds, their initial size, and
     their growth rate are determined by static constants called MANY_WEEDS,
     WEED_SIZE, and WEED_RATE.
  2. Create a bunch of new herbivores, and put the references to these new her-
     bivores in a Vector called fish. The number of fish and their initial size
     are determined by static constants INIT_FISH and FISH_SIZE. In this
     simple simulation, the fish will not grow (their growth rate is zero), and
     their weekly need will be their initial size times a static constant called
     FRACTION.
  3. For each week of the simulation, we will first cause some randomly
     selected fish to nibble on randomly selected weeds. On average, each fish
     will nibble on AVERAGE_NIBBLES weeds (where AVERAGE_NIBBLES is yet
     another constant in our program). After all these nibbles, we will activate
     simulateWeek for each fish and each weed. Dead fish will be removed
     from the fish Vector. At the end of the week, we will give birth to some
     new fish. The total number of newly spawned fish is the current number
     of fish times a constant called BIRTH_RATE. To simplify the simulation,
     we will have the new fish born fully grown with a growth rate of zero.
```

At the end of each week (simulated in Step 3), our program prints a few statistics. These statistics show the number of fish that are currently alive and the total mass of the weeds.

Our program implementing the pseudocode is given in Figure 13.10. The top of the program lists the various constants that we have mentioned, from MANY_WEEDS to BIRTH_RATE. Within the program, we use two static methods to carry out some subtasks. One of the methods, called pondWeek, carries out the simulation of one week in the pond, as described in Step 3 of the pseudocode. The other method, totalMass, computes the total mass of all the plants in the pond.

We discuss some implementation details starting on page 648.

FIGURE 13.10 The Pond Life Simulation

Java Application Program

```
// FILE: PondLife.java
// A simple simulation program to model the fish and weeds in a pond

import edu.colorado.simulations.*; // Provides Organism, Plant, Herbivore classes
import java.util.Vector;

public class PondLife
{
   // Number of weeds in the pond
   public static final int MANY_WEEDS = 2000;

   // Initial size of each weed, in ounces
   public static final double WEED_SIZE = 15;

   // Growth rate of weeds, in ounces/week
   public static final double WEED_RATE = 2.5;

   // Initial number of fish in the pond
   public static final int INIT_FISH = 300;

   // Fish size, in ounces
   public static final double FISH_SIZE = 50;

   // A fish must eat FRACTION times its size during one week, or it will die.
   public static final double FRACTION = 0.5;

   // Average number of weeds nibbled by a fish over a week
   public static final int AVERAGE_NIBBLES = 30;

   // At the end of each week, some fish have babies. The total number of new fish born is the
   // current number of fish times the BIRTH_RATE (rounded down to an integer).
   public static final double BIRTH_RATE = 0.05;

   // Number of weeks to simulate
   public static final int MANY_WEEKS  = 38;
```

(continued)

(FIGURE 13.10 continued)

```
public static void main(String[ ] args)
{
   Vector fish = new Vector(INIT_FISH);    // A Vector of our fish
   Vector weeds = new Vector(MANY_WEEDS); // A Vector of our weeds
   int i;                                  // Loop control variable

   // Initialize the Vectors of fish and weeds
   for (i = 0; i < INIT_FISH; i++)
      fish.addElement(new Herbivore(FISH_SIZE, 0, FISH_SIZE * FRACTION));
   for (i = 0; i < MANY_WEEDS; i++)
      weeds.addElement(new Plant(WEED_SIZE, WEED_RATE));

   // Print headings for the output, using tabs (\t) to separate columns of data
   System.out.println("Week \tNumber \tPlant Mass");
   System.out.println("     \tof     \t(in ounces)");
   System.out.println("     \tFish");

   // Simulate the weeks
   for (i = 1; i <= MANY_WEEKS; i++)
   {
      pondWeek(fish, weeds);
      System.out.print(i + "\t");
      System.out.print(fish.size( ) + "\t");
      System.out.print((int) totalMass(weeds) + "\n");
   }
}

public static double totalMass(Vector organisms)
{
   int i;
   double answer = 0;
   Organism next;

   for (i = 0; i < organisms.size( ); i++)
   {
      next = (Organism) organisms.elementAt(i);
      if (next != null)
         answer += next.getSize( );
   }
   return answer;
}
```

(continued)

(FIGURE 13.10 continued)

```
public static void pondWeek(Vector fish, Vector weeds)
{
      int i;
      int manyIterations;
      int index;
      Herbivore nextFish;
      Plant nextWeed;

      // Have randomly selected fish nibble on randomly selected plants
      manyIterations = AVERAGE_NIBBLES * fish.size( );
      for (i = 0; i < manyIterations; i++)
      {
         index = (int) (Math.random( ) * fish.size( ));
         nextFish = (Herbivore) fish.elementAt(index);
         index = (int) (Math.random( ) * weeds.size( ));
         nextWeed = (Plant) weeds.elementAt(index);
         nextFish.nibble(nextWeed);
      }

      // Simulate the weeks for the fish
      i = 0;
      while (i < fish.size( ))
      {
         nextFish = (Herbivore) fish.elementAt(i);
         nextFish.simulateWeek( );
         if (nextFish.isAlive( ))
            i++;
         else
            fish.removeElementAt(i);
      }

      // Simulate the weeks for the weeds
      for (i = 0; i < weeds.size( ); i++)
      {
         nextWeed = (Plant) weeds.elementAt(i);
         nextWeed.simulateWeek( );
      }

      // Create some new fish, according to the BIRTH_RATE constant
      manyIterations = (int) (BIRTH_RATE * fish.size( ));
      for (i = 0; i < manyIterations; i++)
          fish.addElement(new Herbivore(FISH_SIZE, 0, FISH_SIZE * FRACTION));
   }
}
```

Pondlife—Implementation Details

The implementations of `totalMass` and `pondWeek` are part of Figure 13.10. The `pondWeek` implementation requires the ability to grab a random element out of a `Vector`, and this is accomplished by generating a random integer via the `Math.random` method.

Both `totalMass` and `pondWeek` require the ability to step through the items of a `Vector` one at a time. This could be accomplished by using an `Iterator` (as discussed in Section 5.4), but we just stepped through the elements one at a time in a for-loop, using the `elementAt` method. For example, within the `pondWeek` method, we activate `simulateWeek` for each plant by using the for-loop, as shown here:

```
// Simulate the weeks for the weeds.
for (i = 0; i < weeds.size( ); i++)
{
   nextWeed = (Plant) weeds.elementAt(i);
   nextWeed.simulateWeek( );
}
```

Notice that the return value from `elementAt` is a Java `Object`, so we need the typecast to assure the compiler that this is an acceptable narrowing conversion to a `Plant`. A similar loop steps through the fish, simulating one week for each fish and removing dead fish.

Using the Pond Model

No doubt you have noticed that our pond model is not entirely rooted in reality. For example, each fish is born full grown and does not continue to grow. Some extensions to make the model more realistic are given in the Programming Projects of this chapter. Nevertheless, even our simple program illustrates the principles of simulation programs. Let's look at one way that a simulation program such as ours could be used.

Suppose that your friend Judy owns a pond with 2000 weeds, about 15 ounces each. And perhaps the pond is too choked with weeds for Judy's taste. One way to control the weeds is to introduce a weed-eating species of fish, and the pond life program can help us predict what will happen when a certain number of fish are put in the pond. For example, suppose we have a species of fish where the program's constants (Figure 13.10 on page 645) are accurate. When we run the program with these constants, we get the output in Figure 13.11 on page 649.

Actually, if you run the program, you may get slightly different output because of the use of the random factor in the selection of which fish nibble which weeds. What does the program predict will happen in the pond if we introduce 300 of this kind of fish? Each output line gives the fish population and the plant mass at the end of one more week. The model predicts that the mass of the weeds will decrease fairly rapidly. This is followed by a period of some oscillation in both the fish and plant populations, including a rather catastrophic week for the fish when their population drops from 359 to 144. Sudden declines such as this are observed in real ecosystems when a species is allowed to expand, limited only by its food supply.

This kind of model can provide predictions and test theories of interactions in an ecosystem. It's also important to remember that any predictions are only as accurate as the underlying model.

FIGURE 13.11 Pond Life Results

Sample Output

```
Week    Number  Plant Mass
        of      (in ounces)
        Fish
1       315     27500
2       330     24625
3       346     21375
4       363     17725
5       379     13654
6       359     9286
7       144     6462
8       109     7960
9       112     10245
10      117     12445
11      122     14520
12      128     16470
13      134     18270
14      140     19920
15      147     21420
16      154     22745
17      161     23895
18      169     24870
19      177     25645
20      185     26220
21      194     26595
22      203     26745
23      213     26670
24      223     26345
25      234     25770
26      245     24920
27      257     23795
28      268     22374
29      281     20674
30      292     18656
31      306     16356
32      313     13720
33      301     10984
34      244     8689
35      189     7812
36      163     8225
37      161     9176
38      164     10159
```

Self-Test Exercises

8. Write code to declare a `Vector`. Put ten new organisms in the `Vector`, with an initial size of 16 ounces and a growth rate of 1 ounce per week. Grab five random organisms, and alter their growth rates to 2 ounces per week. Finally, calculate the total of all the organisms' growth rates, and print the result.

9. In the previous exercise, you started with ten organisms growing at 1 ounce per week. Five random organisms had their growth rates changed to 2 ounces per week, so you might think that the total of all the organisms' rates would be `5*1 + 5*2`, which is `15`. But when I ran the program, the total was only `14`. Why?

10. What advantages did we get by storing the fish and weeds in vectors rather than in a partially filled array?

11. Design and implement a new class extended from the `Animal` class. The new class, called `Carnivore`, has one new method with the heading shown here:

```
public void chase(Animal prey, double chance)
```

When `chase(prey, chance)` is activated for some carnivore, the carnivore chases the prey. The probability of actually catching the prey is given by the parameter `chance` (which should lie between 0 and 1—for example 0.75 for a 75% chance). If the prey is actually caught, then this will also activate the carnivore's `eat` method and (sadly) activate the prey's `expire` method.

Note: You can use the `Math.random` method to determine whether the animal is caught, as shown here:

```
if (Math.random( ) < chance)
{
    || Code for catching and eating the prey
}
```

13.3 USING EXTENDED CLASSES FOR ADTS

Many ADTs from previous chapters can be implemented as extended classes, resulting in less programming and less debugging—an all around good way to reuse previous work. For example, consider the stack ADT from Section 6.1. Many of the stack operations are similar to the sequence ADT that you first implemented in Section 3.3, and later revised in Section 4.5.

Now suppose that you have implemented a sequence class with documentation shown in Figure 13.12. This version of the sequence class is implemented with a linked list, and it stores a collection of Java objects. We don't really care about the details of the implementation—in fact, perhaps some other programmer implemented this sequence, and we don't have access to the implementation details.

Our question is this: *Is there some way that the sequence class can be used to quickly implement a new stack class?* Yes. A new stack class can be implemented as an extended class, using the `LinkedSequence` class as the superclass. The plan is for the stack to push and pop items from only one end of the list, ignoring many of the other features of a sequence.

FIGURE 13.12 Documentation for a Sequence Class

Class LinkedSeq

❖ **public class LinkedSeq from the package edu.colorado.collections**

A `LinkedSeq` is a sequence of references to Objects. The sequence can have a special "current element," which is specified and accessed through four methods that are not available in the bag classes (`start`, `getCurrent`, `advance` and `isCurrent`).

Limitations:
Beyond `Int.MAX_VALUE` elements, the `size` method does not work.

Specification

◆ **Constructor for the LinkedSeq**

`public LinkedSeq( )`

Initialize an empty sequence.

Postcondition:
This sequence is empty.

◆ **addAfter and addBefore**

```
public void addAfter(Object element)
public void addBefore(Object element)
```

Adds a new element to this sequence, either before or after the current element.

Parameters:
`element` – a reference to the new element that is being added

Postcondition:
A reference to the element has been added to this sequence. If there was a current element, then `addAfter` places the new element after the current element and `addBefore` places the new element before the current element. If there was no current element, then `addAfter` places the new element at the end of the sequence and `addBefore` places the new element at the front of the sequence. In all cases, the new element becomes the new current element of the sequence. Note that the newly added element may be a null reference.

Throws: `OutOfMemoryError`
Indicates insufficient memory to increase the size of this sequence.

(continued)

(FIGURE 13.12 continued)

◆ **addAll**

`public void addAll(LinkedSeq addend)`

Place the contents of another sequence at the end of this sequence.

Parameters:
`addend` – a sequence whose contents will be placed at the end of this sequence

Precondition:
The parameter, `addend`, is not null.

Postcondition:
The elements from `addend` have been placed at the end of this sequence. The current element of this sequence remains where it was, and the `addend` is also unchanged.

Throws: `NullPointerException`
Indicates that `addend` is null.

Throws: `OutOfMemoryError`
Indicates insufficient memory to increase the size of this sequence.

◆ **advance**

`public void advance( )`

Move forward, so that the current element is now the next element in the sequence.

Precondition:
`isCurrent( )` returns `true`.

Postcondition:
If the current element was already the end element of the sequence (with nothing after it), then there is no longer any current element. Otherwise, the new element is the element immediately after the original current element.

Throws: `IllegalStateException`
Indicates that there is no current element, so `advance` may not be activated.

◆ **clone**

`public Object clone( )`

Generate a copy of this sequence.

Returns:
The return value is a copy of this sequence. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be type cast to a `LinkedSeq` before it is used.

Throws: `OutOfMemoryError`
Indicates insufficient memory for creating the clone.

(continued)

(FIGURE 13.12 continued)

- **concatenation**

  `public static LinkedSeq concatenation(LinkedSeq s1, LinkedSeq s2)`
  Create a new sequence that contains all the elements from one sequence followed by another.

  Parameters:
  `s1` and `s2` – two sequences

  Precondition:
  Neither `s1` nor `s2` is null.

  Returns:
  a new sequence that has the elements of `s1` followed by `s2` (with no current element)

  Throws: `NullPointerException`
  Indicates that one of the arguments is null.

  Throws: `OutOfMemoryError`
  Indicates insufficient memory for the new sequence.

- **getCurrent**

  `public Object getCurrent( )`
  Accessor method to determine the current element of the sequence.

  Precondition:
  `isCurrent( )` returns `true`.

  Returns:
  the current element of the sequence

  Throws: `IllegalStateException`
  Indicates that there is no current element.

- **isCurrent**

  `public boolean isCurrent( )`
  Accessor method to determine whether this sequence has a specified current element that can be retrieved with the `getCurrent` method.

  Returns:
  `true` (there is a current element) or `false` (there is no current element at the moment)

- **removeCurrent**

  `public boolean removeCurrent( )`
  Remove the current element from this sequence.

  Precondition:
  `isCurrent( )` returns `true`.

  Postcondition:
  The current element has been removed from the sequence, and the following element (if there is one) is now the new current element. If there was no following element, then there is now no current element.

  Throws: `IllegalStateException`
  Indicates that there is no current element, so `removeCurrent` may not be activated.

(continued)

(FIGURE 13.12 continued)

◆ **size**

`public long size( )`

Accessor method to determine the number of elements in this sequence.

Returns:
the number of elements in this sequence

◆ **start**

`public void start( )`

Set the current element at the front of the sequence.

Postcondition:
The front element of this sequence is now the current element (but if the sequence has no elements at all, then there is no current element).

Pushing and Popping for the Extended Stack Class

Our new stack class will extend the `LinkedSeq` class from Figure 13.12. The extension will add new methods so that a programmer can use the extended class as if it were a stack. A programmer can also use our new stack as if it were an object of the superclass, `LinkedSeq`.

Our plan is to have the extended class act like a stack by "pushing" and "popping" elements from the head of the sequence that is maintained by the superclass. Also, each time a push or pop occurs, the "current element" of the sequence will be set to the front of the sequence. With this plan, the push and pop of the extended can be implemented as shown here:

```
public void push(Object element)
{
   start( );
   addBefore(element);
}

public Object pop( )
{
   Object answer;

   if (size( ) < 0)
      throw new EmptyStackException( );
   start( );
   answer = getCurrent( );
   removeCurrent( );
   return answer;
}
```

Notice that within the implementations, the methods of the superclass can be activated just like any other method.

The documentation and implementation for the new stack class are shown in Figure 13.13. We have also included the stack's `isEmpty` and `peek` methods, so that the resulting stack has all the important stack methods from Figure 6.1 on page 281. The approach we have taken is similar to Java's own stack (`java.util.Stack`), which is extended from `java.util.Vector`. There are two other points that you should notice about the new stack, involving the constructor and the `clone` method.

The constructor. Notice that the new stack does not declare its own constructor. Therefore, it can automatically use the no-arguments constructor of its superclass. For example, a program can make the declaration

```
DerivedStack s = new DerivedStack( );
```

Cloning a `DerivedStack`. We did not implement a `clone` method for the `DerivedStack`. However, the superclass, `LinkedSeq`, implements the `Cloneable` interface, and it has a `clone` method. Therefore, the extended class also inherits the `clone` method, and we can clone a `DerivedStack` object. With this example in mind, we can now explain the pattern that must be used for every clone method that you implement. The pattern, introduced in Chapter 2 on page 78, looks like this:

```
public Object clone( )
{
   Location answer;

   try
   {
      answer = (Location) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {
      throw new InternalError(e.toString( ));
   }
   ...other work that needs to modify the answer...
```

When we write all of our `clone` methods with this pattern, each clone activation will eventually activate the `clone` method from Java's `Object` class. And this clone method always creates an object of the correct type (of the same type as the object that activated `clone` in the first place). Therefore, if `s` is a `DerivedStack`, then `s.clone( )` will return an object that is really a `DerivedStack` (and not just a `LinkedSeq` object).

FIGURE 13.13 The Stack Class Derived from LinkedSeq

Class DerivedStack

❖ **public class DerivedStack from the package edu.colorado.collections**
➤ **extends LinkedSeq**

A `DerivedStack` is a `LinkedSeq` that can easily be used as if it were a stack of Objects.

Specification

In addition to the `LinkedSeq` methods, a `DerivedStack` object has:

- **isEmpty**

  `public boolean isEmpty( )`

  Determine whether this stack is empty.

  Returns:
  `true` if this stack is empty; otherwise `false`.

- **peek**

  `public Object peek( )`

  Get the top item of this stack, without removing the item.

  Precondition:
  This stack is not empty.

  Returns:
  the top element of the stack (and sets the current element to the head of the sequence)

  Throws: `EmptyStackException`
  Indicates that this stack is empty.

- **pop**

  `public Object pop( )`

  Get the top element, removing it from this stack.

  Precondition:
  This stack is not empty.

  Postcondition:
  The return value is the top item of this stack, and the element has been removed. The new current element of the sequence is the element that used to be second.

  Throws: `EmptyStackException`
  Indicates that this stack is empty.

- **push**

  `public void push(Object element)`

  Push a new item onto this stack.

  Parameters:
  `item` – the item to be pushed onto this stack

  Postcondition:
  The item has been pushed onto this stack (and it is now the current element of the sequence)

  Throws: `OutOfMemoryException`
  Indicates insufficient memory for pushing a new item onto this stack.

(continued)

(FIGURE 13.13 continued)

Implementation

```
// FILE: DerivedStack.java from the package edu.colorado.geometry
// Documentation is available on the previous page or from the DerivedStack link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;
import java.util.EmptyStackException;

public class DerivedStack extends LinkedSeq
{
   public boolean isEmpty( )
   {
      return (size( ) == 0);
   }

   public Object peek( )
   {
      if (size( ) < 0)
         throw new EmptyStackException( );
      start( );
      return getCurrent( );
   }

   public Object pop( )
   {
      Object answer;

      if (size( ) < 0)
         throw new EmptyStackException( );
      start( );
      answer = getCurrent( );
      removeCurrent( );
      return answer;
   }

   public void push(Object element)
   {
      start( );
      addBefore(element);
   }

}
```

Self-Test Exercises

12. Our original stack from Chapter 6 has a `size` method. Does a `DerivedStack` have a `size` method?
13. Suppose that you are implementing an extended class called `Tribble`. The superclass, called `Willis`, implements the `Cloneable` interface and `Tribble` does not override the `clone` method. If `t` is a `Tribble`, then what is the actual data type of the `Object` returned by `t.clone( )`? (To read more about Willis, you'll have to track down *Red Planet* by Robert A. Heinlein.)
14. Implement a new method of the stack class. The method has one parameter, `i`, and the precondition requires the stack to contain at least `i` elements. The method returns the element that is `i` positions from the top of the stack, and it sets the current element of the sequence to this element. For example, an argument of 1 will return the top element, an argument of 2 returns the element under the top, and so on.
15. Which would be easier: to implement a bag as an extended class of the sequence, or to implement the sequence as an extended class of the bag?

CHAPTER SUMMARY

- Object-oriented programming supports the use of *reusable components* by permitting new *extended classes* to be declared that automatically *inherit* all members of an existing superclass.
- All members of a superclass are inherited by the extended class, but only the nonprivate members of the superclass can be accessed by the programmer who implements the extended class. This is why most of our examples of superclasses do not specify the precise form of the private members of the superclass.
- The connection between an extended class and its superclass can often be characterized by the *is-a* relationship. For example, an herbivore *is-a* particular kind of animal, so it makes sense to implement `Herbivore` as an extended class of the `Animal` superclass.
- When you implement a new class, ask yourself whether the new class is an example of an existing class with slightly different capabilities or extra capabilities. In these cases, the new class can be implemented as an extended class of the existing class.

FURTHER READING

This chapter has introduced the concept of extended classes and inheritance, which is a central concept for OOP programming. In your future programming, further inheritance is likely to be important. You can consult a comprehensive Java language guide such as *Just Java and Beyond* by Peter van der Linden.

Solutions to Self-Test Exercises ?

1. The declaration for the extended class is shown here, along with the implementation of the new method:

```
public class DaylightClock
extends Clock
{
   public boolean isDay( )
   {
      if (isMorning( ))
         return (getHour( ) >= 7);
      else
         return (getHour( ) < 7);
   }
}
```

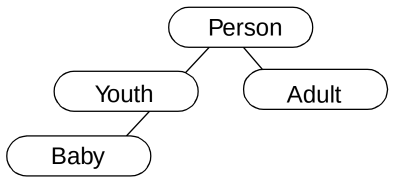
2. Suppose `d` is a `DayLightClock` and `c` is a `Clock`. The assignment `c = d` is a widening conversion. An example of a narrowing conversion is `d = (DayLightClock) c`.

3. It inherits the no-arguments constructor of the superclass.

4. The `NoonAlarm` overrides the `advance` method of the ordinary clock. Here is one solution:

```
public class NoonAlarm extends Clock
{
   public void advance(int minutes)
   {
      int untilNoon;

      // Calculate number of minutes until
      // noon.
      if (isMorning( ))
         untilNoon =
            60 * (12-getHour( ))
            - getMinute( );
      else if (getHour( ) != 12)
         untilNoon =
            60 * (24 - getHour( ))
            - getMinute( );
      else
         untilNoon =
            60 * 24
            - getMinute( );

      // Maybe print an alarm message.
      if (minutes >= untilNoon)
         System.out.print("!!");

      // Activate the superclass method.
      super.advance(minutes);
   }
}
```

5.



6. The `Plant` declaration is:

```
public class Plant extends Organism
{
   public Plant
   (double initSize, double initRate)
   {
      super(initSize, initRate);
   }

   public void
   NibbledOn(double amount)
   {
      if (amount < 0)
         throw new
         IllegalArgumentException
         ("amount is negative");
      if (amount > getSize( ))
         throw new
         IllegalArgumentException
         ("amount is too big");
      alterSize(-amount);
   }
}
```

7. See the solution in Figure 13.10 on page 646. Another approach would be to use the vector's iterator.

8. Here is one solution:

```
Vector blob = new Vector(10);
int i;
double answer;
Organism thing;

for (i = 1; i <= 10; i++)
  blobs.addElement
  (new Organism(16, 1));

for (i = 1; i <= 5; i++)
{
  thing = (Organism)
    blobs.elementAt
    ((int) Math.random( ) * 10);
  thing.setRate(2);
}

answer = 0;
for (i = 0; i <= 10; i++)
{
  thing =
    (Organism) blobs.elementAt(i);
  answer += things.getRate( );
}

System.out.println
(answer + " total of rates");
```

9. The random method must have selected one organism twice (and three other organisms were selected once each).

10. We didn't have to worry about the number of elements going beyond the size of the array. We also get to use various methods such as `removeElementAt`.

11. We'll leave some of this to you, but here is most of the new method:

```
public void chase
(Animal prey, double chance)
{
   ...check that chance is in 0...1
   if (Math.random() < chance)
   {
      eat(prey.getRate( ));
      prey.expire( );
   }
}
```

12. Yes. The `size` method is inherited.

13. `Tribble`

14. Here is one implementation:

```
public Object get(int i)
{
    int count;
    Item answer;

    start( );
    for (count=1; count<i; count++)
        advance( );

    answer = current( );

    return answer;
}
```

15. The easy task is to implement a bag as an extended class, with the sequence as the superclass.

PROGRAMMING PROJECTS

1 A *set* is like a *bag*, except that a set does not allow multiple copies of any element. If you try to insert a new copy of an item that is already present in a set, then the set simply remains unchanged. For this project, implement a set as a new class that is extended from one of your bags.

2 Rewrite the pond life program from Figure 13.10 on page 645 so that the values declared at the start of the program are no longer constant. The program's user should be able to enter values for all of these constants. Also, improve the program so that the fish are more realistic. In particular, the fish should be born at a small size and grow to some maximum size. Each fish should also have a weekly food requirement that is proportional to its current size.

3 Rewrite the pond life program from Figure 13.10 on page 645 so that the output is presented as a graph in an applet window. The applet can use most of the same graphing techniques as the fractal applet from page 388 in Chapter 8. You should use different colors for graphing the populations of the weeds and the fish.

4 Extend the `Organism` object hierarchy from Section 13.2 so that there is a new class `Carnivore` as described in Self-Test Exercise 11 on page 650. Use the hierarchy in a model of life on a small island that contains shrubs, geese that eat the shrubs, and foxes that eat the geese. The program should allow the user to vary the initial conditions on the island (such as number of foxes, the amount of food needed to sustain a fox, and so on).

5 Implement a bag class as an extended class of a seqeunce. Use the sequence from Figure 13.12 on page 651.