Chapter 12 Polymorphism

Chapter Objectives

After studying this chapter, you should be able to:

- Write a polymorphic program using inheritance
- Write a polymorphic program using an interface
- Build an inheritance hierarchy
- Use the Strategy and Factory Method patterns to make your programs more flexible
- Override standard methods in the `Object` class

In science fiction movies, an alien sometimes morphs from one shape to another, as the need arises. Someone shaped like a man may reshape himself into a hawk or a panther or even a liquid. Later, after using the advantages the new shape gives him, he changes back into his original shape.

Morph is a Greek word that means "shape." The prefix *poly* means "many." Thus, *polymorph* means "many shapes." The movie alien is truly polymorphic. However, even though he has many outward shapes, the core of his being remains unchanged.

Java is also polymorphic. A class representing a core idea can morph in different ways via its subclasses. After studying inheritance in Chapter 2, this may sound like nothing new. However, in that chapter, we usually added new methods to a subclass. In this chapter, we will focus much more on overriding methods from the superclass. The power of this technique will become evident when we are free from knowing whether we're using the superclass or one of its subclasses.

We will also find similar benefits in using interfaces.

12.1 Introduction to Polymorphism

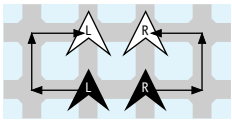
Java provides two ways to implement polymorphism. One uses inheritance and the other uses interfaces. Both depend on having two or more classes that either extend the same class or implement the same interface. We will return to the robot world to illustrate the core ideas and then move to other examples.

KEY IDEA

Polymorphism can be implemented with inheritance or interfaces.

12.1.1 Dancing Robots

Let's define two rather fanciful robots that dance, one to the left and one to the right, as they move to the next intersection. The arrows in Figure 12-1 show the paths they take as they move from their initial position (shown in black) to their final position (shown in white). `LeftDancer` is labeled with an "L" and `RightDancer` is labeled with an "R".



(figure 12-1)

Paths that dancing robots take as they move to the next intersection

The code implementing `LeftDancer` is shown in Listing 12-1. `RightDancer` is similar.

Listing 12-1: *A robot that dances to the left as it moves forward*

```
1  import becker.robots.*;
2
3  /** LeftDancers dance to the left as they move forward.
4  *
5  *  @author Byron Weber Becker */
6  public class LeftDancer extends RobotSE
7  {
8    public LeftDancer(City c, int str, int ave, Direction dir)
9    { super(c, str, ave, dir);
10     this.setLabel("L");
11   }
12
13   /** Dance to the left. */
14   public void move()
15   { this.turnLeft();
16     super.move();
17     this.turnRight();
```

FIND THE CODE

ch12/dancers/

Listing 12-1: *A robot that dances to the left as it moves forward* (continued)

```
18      super.move();
19      this.turnRight();
20      super.move();
21      this.turnLeft();
22    }
23  }
```

Method Resolution Review

How Java determines which method to execute is called method resolution. This concept was first discussed in Section 2.6.2, but it is worth reviewing because it is important to understand how these classes work.

When a method is invoked, say `karel.move()`, Java looks for the `move` method beginning with the object's class. If the object was originally created with the phrase `new LeftDancer(...)`, Java will look for the `move` method beginning with the `LeftDancer` class. That class has a `move` method, so it's executed.

On the other hand, suppose that the `turnLeft` method was invoked. Once again, the search for the method begins with the object's class, `LeftDancer`. That class, however, doesn't have a `turnLeft` method. The search continues in its superclass, `RobotSE`. It doesn't have a `turnLeft` method either and so the search continues in its superclass. `Robot` has a `turnLeft` method; that's the method that is executed. The search for the method to execute starts with the object's actual class and proceeds up the inheritance hierarchy until it is found. If no such method exists, that fact is determined when the program is compiled and an error message is issued.

LOOKING AHEAD

What would happen if line 16 was `this.move()`*? See Written Exercise 12.1.*

When the statement uses `super` to call the method, the search starts at a different place—the superclass of the class containing the method. Thus, the statement `super.move()` at lines 16, 18, and 20 in Listing 12-1 begins to search for `move` in the `RobotSE` class, executing the first `move` method found as it moves up the inheritance hierarchy. In this case, it executes the `move` method in the `Robot` class, resulting in the familiar movement from one intersection to another.

12.1.2 Polymorphism via Inheritance

So far we haven't seen anything new. `LeftDancer` could have been an assignment in Chapter 2. So where is the polymorphism? It's in how these classes are *used*.

Let's use these classes in a way that appears silly at first: Let's assign a `LeftDancer` to a `RobotSE` reference variable, as follows:

```
RobotSE karel = new LeftDancer(...);
```

Java allows this kind of assignment, as long as the reference on the right is a subclass of the reference on the left. It would not work to assign a `LeftDancer` to a `City` variable or even to a `RightDancer` variable because neither is a superclass of `LeftDancer`.

If we can do this, we can also put several `LeftDancers` and `RightDancers` into a single array. Imagine a chorus line of dancing robots, as implemented in Listing 12-2. The core feature is an array that contains *all* the robots, no matter what their type.

Listing 12-2: *An array filled with different kinds of robots*

```
1  import becker.robots.*;
2
3  /** Run a chorus line of dancing robots.
4  *
5  *  @author Byron Weber Becker */
6  public class DanceHall
7  {
8    public static void main(String[] args)
9    { City stage = new City();
10     RobotSE[] chorusline = new RobotSE[5];
11
12     // Initialize the array.
13     chorusline[0] = new LeftDancer(
14                              stage, 1, 0, Direction.EAST);
15     chorusline[1] = new RightDancer(
16                              stage, 2, 0, Direction.EAST);
17     chorusline[2] = new LeftDancer(
18                              stage, 3, 0, Direction.EAST);
19     chorusline[3] = new RightDancer(
20                              stage, 4, 0, Direction.EAST);
21     chorusline[4] = new RobotSE(
22                              stage, 5, 0, Direction.EAST);
23
24     for (int i = 0; i < chorusline.length; i++)
25     { chorusline[i].move();
26     }
27   }
28 }
```

FIND THE CODE

ch12/dancers/

PATTERN

Polymorphic Call

For now, remember that all of the objects in the array have a `move` method. We can tell each of the robots to move with the loop in lines 24–26. But how do these robots move? Do they move like instances of `RobotSE` because the array is declared that way, or do they each move like the `LeftDancer`, `RightDancer`, or `RobotSE` that they really are?

KEY IDEA

Polymorphism uses a subclass as if it were a superclass, relying on the subclass to override methods appropriately.

The answer is that each object executes the `move` method in its own class. That is, a `LeftDancer` moves to the left because that's how that kind of robot was defined to move. `RightDancers` move to the right, as their `move` method says they should. The lone `RobotSE` at the end of the line moves as any other instance of `RobotSE` would move.

This is polymorphism in action: the statement `chorusline[i].move()` tells a robot to move, but this particular statement does not need to know or care what kind of robot it is. For example, it doesn't need to tell the `LeftDancers` to move to the left. It just tells each robot to move and that robot moves in the way it is defined to move. This is like a choreographer telling a dance troupe to "begin on the count of three: one, two, three." All the dancers begin dancing their parts without individual instruction from the choreographer.

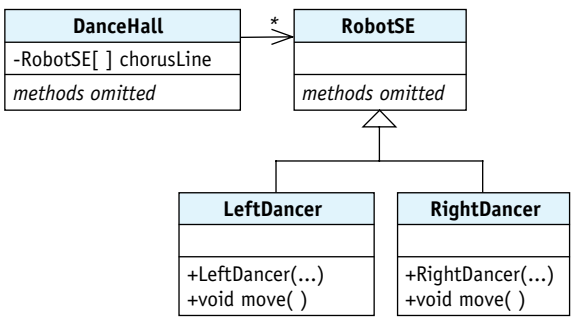
KEY IDEA

Polymorphic programs have key identifying features.

A class diagram for what we have just done is typical of polymorphic programs and is shown in Figure 12-2. The characteristic feature is a superclass (`RobotSE`) that is extended with at least two subclasses. Another class—`DanceHall` in this case—uses instances of the subclasses as if they were the superclass.

(figure 12-2)

Class diagram for a polymorphic robot program



Adding a New Method

Consider adding a `pirouette` method to both `LeftDancer` and `RightDancer`. When a dancer pirouettes, she turns completely around. A `LeftDancer` turns to the left, as follows, whereas a `RightDancer` turns to the right.

```
public class LeftDancer extends RobotSE
```

```
{  // Constructor and move method omitted.

   /** Turn completely around. */
   public void pirouette()
   { this.turnLeft();
     this.turnLeft();
     this.turnLeft();
     this.turnLeft();
   }
}
```

FIND THE CODE

ch12/dancers2/

With this change, can we tell the dancers in `chorusline` to pirouette?

```
1  RobotSE[] chorusline = new RobotSE[5];
2  // Initialization of chorusline is omitted.
3  for (int i = 0; i < chorusline.length; i++)
4  { chorusline[i].pirouette();
5  }
```

We cannot. This code will not even compile because line 1 declares that each element of `chorusline` will refer to a `RobotSE` object or one of its subclasses. Most kinds of robots do not have a `pirouette` method and so the compiler assumes the worst—that in line 4, `chorusline[i]` refers to an ordinary robot that lacks a `pirouette` method.

KEY IDEA

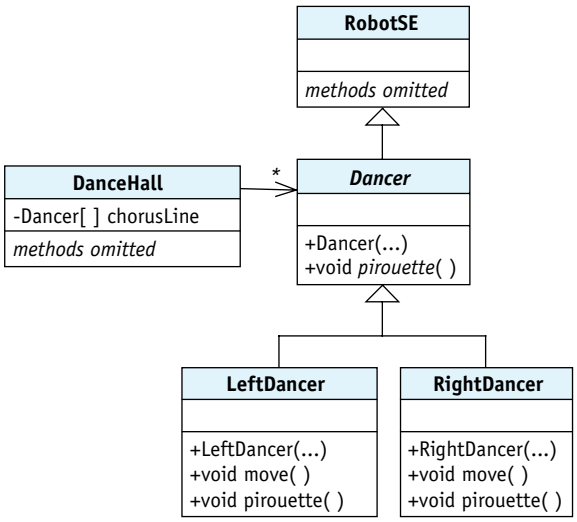
The reference's type determines which methods can be called; the object's type determines which code is executed.

The rule is this: The type of the reference variable determines the names of the methods that can be called; the type of the actual object determines which code is executed. In this example, `chorusline[i]` is the reference variable and its type is `RobotSE`. Therefore, the only methods you can call are methods that appear in the `RobotSE` class. On the other hand, when you call one of those methods (like `chorusline[i].move()`), the type of the actual object (for example, `LeftDancer`) is what determines how the robot moves.

To include the `pirouette` method in a dancer's repertoire, we need to add a new class, as shown in Figure 12-3. The `Dancer` class extends `RobotSE` and adds a `pirouette` method. The `DanceHall` class is changed to use an array of `Dancer` objects rather than `RobotSE`. This implies that the single `RobotSE` object shown at line 17 of Listing 12-2 can no longer be included in the array because it is not a subclass of `Dancer`.

(figure 12-3)

Using an abstract class; abstract classes and methods are labeled in italics



Abstract Classes

When we write the code for the `Dancer` class, should `pirouette` turn to the left like a `LeftDancer` or turn to the right like a `RightDancer`? No matter which choice we make, it will be wrong for at least one of the subclasses.

The best option is to make `pirouette` an **abstract method**. Such a method includes only the access modifier, return type, and signature (method name and parameter list). The method body is replaced with a semicolon. For example:

```
/** Turn this dancer around 360 degrees in its preferred direction. */
public abstract void pirouette();
```

KEY IDEA

An abstract method enables polymorphism even when implementation details are not known.

The purpose of an abstract method is to declare a name that can be used polymorphically, even though it does not declare how the method will be implemented.

Abstract methods must be overridden in a subclass to supply a method body. For example, `pirouette` is overridden in `LeftDancer` with a method that turns to the left; in `RightDancer`, it is overridden with a method that turns to the right.

A class that declares or inherits a method without a body is called an **abstract class** and must be declared with the keyword `abstract`, as follows:

```
public abstract class Dancer extends RobotSE
```

An abstract class such as `Dancer` can be extended by another class, *`X`*, even though *`X`* does not supply a body for `pirouette`. However, *`X`* must also be declared abstract.

An abstract class cannot be used to instantiate an object.

When an abstract method or class is shown in a class diagram, its name will be in italics, as shown in Figure 12-3.

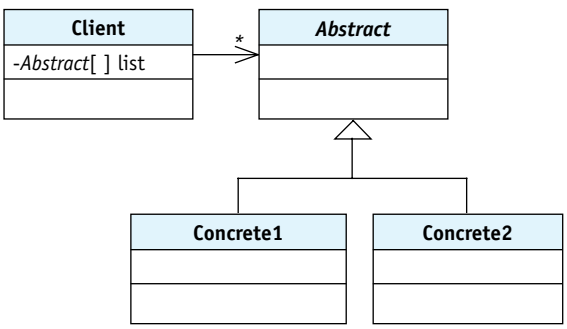
KEY IDEA

Names of abstract methods and classes are shown in italics in class diagrams.

12.1.3 Examples of Polymorphism

Let's take a brief break from robots to examine a number of other examples where polymorphism may be appropriate. All of these cases have the same basic structure as the `DanceHall` example shown in Figure 12-2. In Figure 12-4, we give the participating classes more general names so that in the examples that follow, we can identify how the classes interact. We will use the names as follows:

- The **client class** uses the services of another class. In the previous example, `DanceHall` is the client that uses the services (`move`) of another class—it just happens to use them polymorphically.
- The **abstract class** is used to declare variables in the client. It also lists the methods that can be used by the client. In Figure 12-2, `RobotSE` is the abstract class; in Figure 12-3, it's `Dancer`. (The class that defines the names used polymorphically is called "abstract" even though it might not use the `abstract` keyword.)
- A **concrete class** implements the methods named in the abstract class. In the previous example, `LeftDancer` and `RightDancer` are both concrete classes.



(figure 12-4)

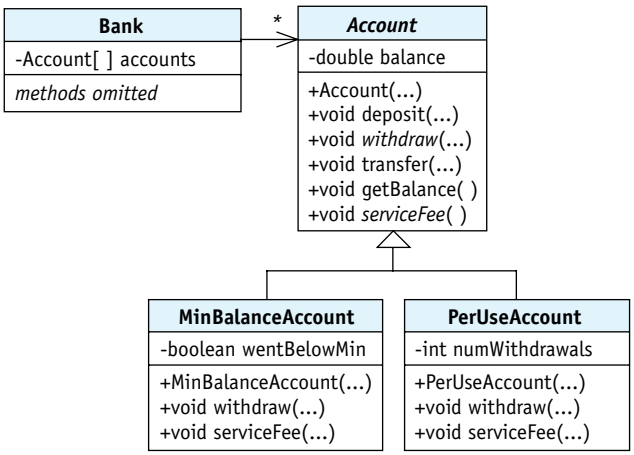
Common pattern for inheritance-based polymorphism

Example: Bank Accounts

A `Bank` class (the client) has many `Accounts` (the abstract class). The `Account` class has both an instance variable to maintain the account's balance and methods to deposit money, withdraw money, and transfer money to another account. It also has methods to get the balance and to charge a service fee at the end of the month. See Figure 12-5.

(figure 12-5)

Class diagram for a bank



However, each account is really an instance of `MinBalanceAccount` or `PerUseAccount`, both concrete classes. A `MinBalanceAccount` is a kind of account that is free as long as the customer maintains a minimum balance of $1,000. The `withdraw` method is overridden to set an instance variable if the balance ever goes below the minimum. The service fee method is also overridden to charge (or not charge) the service fee.

Similarly, `PerUseAccount` overrides the `withdraw` method to count the number of withdrawals. It also overrides the `serviceFee` method to charge the appropriate fee based on the number of withdrawals.

Polymorphic Call

With this design, the `Bank` class can process every transaction in the same way. It doesn't need to know or care what kind of account the customer has because each account will handle the transaction in a manner that is appropriate for that account.

Example: Drawing Program

A drawing program constructs a drawing out of different kinds of shapes: ovals, rectangles, lines, polygons, characters, and so on. In this case, `Drawing` would be the client class. It has an array of `Shape` objects. `Shape` is the abstract class. It's most crucial method is `draw`.

Classes like `Oval`, `Rectangle`, and `Line` are the concrete classes that extend `Shape`. Each of them override the `draw` method to draw the appropriate shape: an `Oval` draws an oval, a `Rectangle` draws a rectangle, and a `Line` draws a line.

With this design, the `Drawing` class can draw the entire image with a simple `for` loop, which tells each `Shape` object in its array to draw itself, as follows:

```
for (int i = 0; i < this.numShapes; i++)
{ this.shapes[i].draw(...);
}
```

PATTERN

Polymorphic Call

Example: Computer-Game Strategies

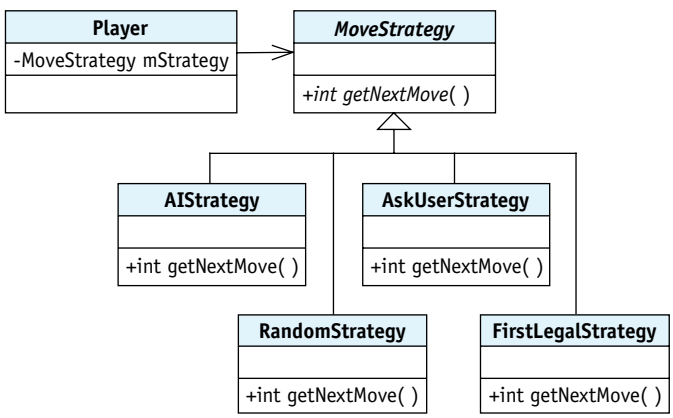
Computer versions of chess, Monopoly, and various card games have two or more players. Sometimes the players are people and sometimes the computer controls the extra players. Sometimes the computer has several skill levels.

Each `Player` object must have a strategy for generating its next move. There might be several ways to do this: ask a human for the next move, find the first legal move, generate a random move, or invoke some sophisticated "artificial intelligence."

The idea of polymorphism is useful here. `Player` is the client class. Rather than an array, it has a single instance variable holding a `MoveStrategy` object. This class is the abstract class shown in Figure 12-4. Its most important method is `getNextMove`. `MoveStrategy` is extended by several concrete classes: `AskUserStrategy`, `FirstLegalStrategy`, `RandomStrategy`, and `AIStrategy`. They each override `getNextMove` to get the next move for the player in their own particular way (see Figure 12-6).

PATTERN

Strategy



(figure 12-6)

Class diagram of a `Player` class that uses a polymorphic move strategy

When the game program is set up this way, a `Player` object can ask for its next move without knowing or caring which particular strategy is being used to generate the move. The strategy can even be changed mid-game by simply assigning a new subclass of `MoveStrategy` to the `Player` object's instance variable.

12.1.4 Polymorphism via Interfaces

In a polymorphic program, the client says *what* it wants done (`move`) but not *how*. How the task is accomplished is determined by the details of the concrete classes.

When polymorphism is achieved via inheritance, the abstract class and the superclass are the same. That combination constrains both what can be done and how it can be implemented. The superclass already contains the methods that can be called, limiting what can be done by the client. The fact that the concrete classes extend the abstract class means that they are not free to extend another class, thus limiting how tasks are accomplished.

KEY IDEA

Java interfaces separate what an object can do from how it can be implemented.

Java interfaces provide another way to implement polymorphism that cleanly separates what can be done from how it can be implemented. Recall from Section 7.6 that interfaces list method signatures and return types, but do not provide the method bodies. For example, the following is an interface for classes that can move:

```
public interface IMove
{
   /** Move this object. */
   public void move();
}
```

We can use this interface with `LeftDancer` and `RightDancer` by including the `implements` keyword and the interface name in the class declaration, as follows:

```
public class LeftDancer extends RobotSE implements IMove
{  // Constructor omitted.
   public void move()
   {  // Same as the move method in Listing 12-1.
   }
}
```

The `implements` clause causes the compiler to verify that `LeftDancer`, or one of its superclasses, implements all of the methods listed in the `IMove` interface.

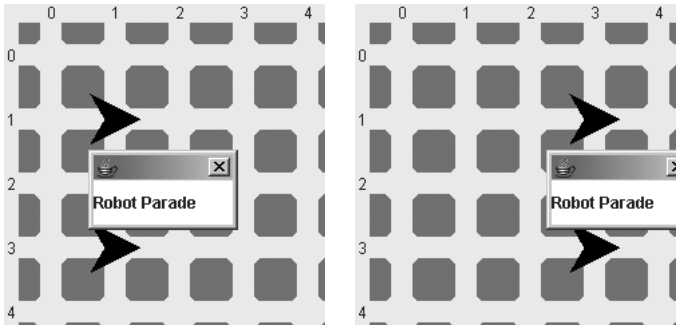
`DanceHall`, the client class, can use the interface to declare the array of dancers, as follows:

```
IMove[] chorusline = new IMove[5];
chorusLine[0] = new LeftDancer(stage, 1, 0, Direction.EAST);
```

However, an instance of `RobotSE` cannot be inserted into the array because it does not implement `IMove`.

It may seem that we haven't gained anything by introducing `IMove`. But imagine a parade of robots where a `LeftDancer` and a `RightDancer` are carrying a banner. We want the banner to float above everything in the city and display the text "Robot

Parade". The banner should move as the robots move. Figure 12-7 shows two screen captures of such a program.



(figure 12-7)

Before and after moving robots and their banner

A class implementing such a banner is shown in Listing 12-3. It displays a small window that floats above all other windows. It has a `move` method to move it a given distance. It extends `JDialog` but also implements the `IMove` interface and can therefore be put in the same array as the robots that carry it, as follows:

```
public static void main(String[] args)
{ City c = new City();

  IMove[] movers = new IMove[3];

  movers[0] = new LeftDancer(c, 1, 1, Direction.EAST);
  movers[1] = new RightDancer(c, 3, 1, Direction.EAST);
  movers[2] = new Banner(80, 165, 40, "Robot Parade");

  for (int numMoves = 0; numMoves < 2; numMoves++)
  { for (int i = 0; i < movers.length; i++)
    { movers[i].move();
    }
  }
}
```

PATTERN

Polymorphic Call

For both the banner and the robots, the client can say *what* to do (`move`), but the details of *how* they move are very different. This is the essence of polymorphism, but using an interface instead of extending a class.

Listing 12-3: *A banner that floats above a city and everything in it*

```
1  import javax.swing.*;
2
3  /** A "banner" that passes over a robot city.
4  *
```

FIND THE CODE

ch12/iMove/

Listing 12-3: *A banner that floats above a city and everything in it* (continued)

```
 * @author Byron Weber Becker */
public class Banner extends JDialog implements IMove
{ private int x;
  private int y;
  private int deltaX;

  /** Display a message in a floating window.
  * @param initX       The initial x position of the banner.
  * @param initY       The initial y position of the banner.
  * @param moveX       The distance to move.
  * @param msg         The msg to display. */
  public Banner(int initX, int initY, int moveX, String msg)
  { super();
    this.deltaX = moveX;
    this.x = initX;
    this.y = initY;
    this.setSize(20, 60);
    this.setLocation(this.x, this.y);
    this.setAlwaysOnTop(true);
    this.setContentPane(new JLabel(msg));
    this.setVisible(true);
  }

  /** Move the banner. */
  public void move()
  { this.x += this.deltaX;
    this.setLocation(this.x, this.y);
  }
}
```

12.1.5 The Substitution Principle

A key to understanding polymorphism is the **substitution principle**, introduced by Barbara Liskov. It says that an object of one type, *A*, can substitute for an object of another type, *B*, if *A* can be used any place that *B* can be used.

For example, consider an automobile rental agency that has vans, sports cars, and sedans. If a customer calls a week ahead to reserve an automobile, the agency can substitute a van if that's what is most available. A van is a kind of automobile and can do everything an automobile can do. On the other hand, if the customer called to reserve

a van, the agency cannot substitute a sports car. Maybe the customer specifically needs the extra passenger space provided by the van.

In Java, a subclass can always be used anywhere the superclass can be used. A `LeftDancer` (the subclass) can always be substituted for a `Dancer` (the superclass)—just like a van (the subclass) can be substituted for an automobile (the superclass). Why? Inheritance guarantees that a `LeftDancer` has all of the methods that a `RobotSE` has.

KEY IDEA

A subclass that does things differently can be substituted for the superclass.

Similarly, a class such as `Banner` can be substituted for its interface because the compiler guarantees that every method named in the interface will be implemented in the concrete class.

KEY IDEA

Classes can be substituted for the interfaces they implement.

A polymorphic program exploits the fact that even though a concrete class may be substituted for the abstract class, they do not necessarily act the same way. The key feature of a polymorphic program is setting up the classes so that some operations can be performed without knowing the actual types of the objects being used.

12.1.6 Choosing between Interfaces and Inheritance

We've seen that a polymorphic program can be written using either interfaces or inheritance. On what basis do we choose one approach over the other?

The simple rule is to use an interface unless there is some commonality between all the concrete classes that can be implemented in a superclass.

KEY IDEA

Use interfaces for polymorphism unless there is a reason to use inheritance.

In the first example, `LeftDancer` and `RightDancer` have many common details that are implemented in `RobotSE` and its superclasses. These include the ability to move the usual way, turn left or right, and display itself in the city. In the second example, there are no such commonalities. The robots and the banner are implemented completely differently with different superclasses—and thus an interface was appropriate.

In Chapter 11 we talked about loose coupling being a good design decision. That is, classes should depend on each other as little as possible. Modern object-oriented design makes extensive use of interfaces to cleanly separate what classes do from how they do them. That is, a client that uses interfaces is less dependent than one that doesn't. If another concrete class becomes available that implements the interface, it can be substituted with no change to the client. That's loose coupling!

12.2 Case Study: Invoices

In Sections 8.3 and 11.2.2 we studied a simple design methodology to help us start writing an object-oriented program. We now extend that methodology for the last time to incorporate polymorphism. The changes from Section 8.3 are shown in italics in Figure 12-8.

(figure 12-8)

Object-oriented design methodology

1. Read the description of what the program is supposed to do, highlighting the nouns and noun phrases. These are the objects your program must declare.
   a. If there are any objects that cannot be directly represented using existing types, define classes to represent such objects.
   b. *If two or more classes have common attributes and pass the 'is-a' test, consolidate those attributes into a superclass, and extend the superclass to define the classes.*
2. Highlight the verbs and verb phrases in the description. These are the services. If a service is not predefined:
   a. Define a method to perform the service.
   b. Place it in the class responsible for providing the service.
   c. *Where necessary, override methods in subclasses.*
   d. *If a class is responsible for a service but cannot implement it, declare an abstract method.*
3. Apply the services from Step 2 to the objects from Step 1 in a way that solves the problem.

In this section, we'll see how this methodology works by applying it to an invoicing application. The problem statement (or specification) and a sample invoice are shown in Figure 12-9.

Print an invoice to request payment for items provided to a customer by the company. The invoice shows the customer's name and address, and the total invoice amount.

In addition to the above, add one line item for each group of identical items sold. Each line item shows the quantity of items sold, a description, the unit cost, and the total amount charged for items in the group.

The company provides three kinds of items:

Goods (like computers or software): calculate the amount charged as the quantity times the unit cost.
Services (such as providing an Internet connection or a service contract on a computer): calculate the amount charged as the quantity (number of connections or contracts) times the unit cost per month times the number of months.
Consulting: calculate the amount charged as the hourly rate times the time spent.

A sample invoice is shown below. Notice that some of the variation between different kinds of items is shown in the description.

Computers To You
1 Byte Way
Waterloo, Ontario N2G 3H4

Byron Weber Becker
122 Nomad Street
Waterloo, Ontario N2L 3G1

Qty	Description	Unit Cost	Amount
3	Desktop computers	$1,750.00	$5,250.00
1	Premium office suite	$750.00	$750.00
3	Computer service contracts (12 months)	$5.95	$214.20
1	Consulting re: printer installation (0.75 hrs)	$75.00	$56.25
1	Consulting re: LAN wiring (5.00 hrs)	$75.00	$375.00
		Total:	$ 6,645.45

(figure 12-9)

Problem statement for a simple invoicing application

12.2.1 Step 1: Identifying Objects and Classes

Step 1 in the object-oriented design methodology (Figure 12-8) tells us to highlight the nouns and noun phrases. Recall that a noun is a word that can refer to a person, place, or thing and is often the subject or object of a verb. The nouns and noun phrases in the problem statement are listed in Figure 12-10 in the left column.

(figure 12-10)

Nouns and noun phrases from the problem statement

Nouns and Noun Phrases	Types	Class Names
invoice		`Invoice`
~~payment for items provided~~		
customer		`Customer`
company		`Company`
customer's name	`String`	
customer's address		`Address`
total invoice amount	`double`	
line item		`LineItem`
~~group of identical items sold~~		
quantity of items sold	`int`	
description of items sold	`String`	
unit cost of items sold	`double`	
total amount charged for items in the group	`double`	
items		`Item`
goods		`Good`
amount charged	`double`	
services		`Service`
unit cost per month	`double`	
number of months	`int`	
consulting		`Consulting`
hourly rate	`double`	
time spent	`double`	

Some of the nouns are not relevant and can be eliminated. For example, "payment for items provided" is in a clause explaining the purpose of the system and represents something the customer does in response to receiving an invoice. Similarly, "group of identical items sold" seems to define the term "line item." These two noun phrases are crossed out in the list.

Some nouns in the list duplicate each other. For example, two entries in the table talk about "unit cost." Furthermore, the sample invoice shows the hourly rate for consulting in the unit cost column. They can probably all be combined into the single term "unit cost."

Some of these nouns can be represented with existing types such as integers and strings. These are noted in the middle column. Other nouns will require that we define a class, as suggested by Step 1a of Figure 12-8. Suggested class names are shown in the right column.

Class Relationships

LOOKING BACK

"Is-a" and "Has-a" are two ways of relating classes. They were discussed in Section 8.1.3.

We've identified a number of potential classes in Figure 12-10. How are they related to each other? If we use the "is-a" and "has-a" tests, the sentence "An invoice has a customer" makes much more sense than "An invoice is a customer." Similarly, "A customer has an address" and "An invoice has a line item" make more sense than saying "A customer is an address" or "An invoice is a line item."

Examining the specification's three paragraphs related to the three kinds of items indicates that `Goods` have amounts charged, quantities, and unit costs. `Services`, on the other hand, have amounts charged, quantities, unit costs per month, and the number of months. Finally, `Consulting` objects have hourly rates and time spent. We see that these classes definitely have some common attributes; therefore, Step 1b of Figure 12-8 (which suggests forming a superclass) may apply. The phrase "three kinds of items" suggests that we might name the superclass `Item` and already hints that inheritance may be appropriate.

The remaining question is whether these classes pass the "is-a" test. Recall that the "is-a" test consists of forming a sentence using "is-a" or "is a kind of" with the two classes in question. For example, "A `Service` is a kind of `Item`" or "A `Consulting` is a kind of `Item`."

These sentences don't sound quite right. The problem might be that the inheritance relationship isn't correct. However, the specification explicitly says that there are "three kinds of items: goods, services, and consulting."

KEY IDEA

Choose appropriate names for classes.

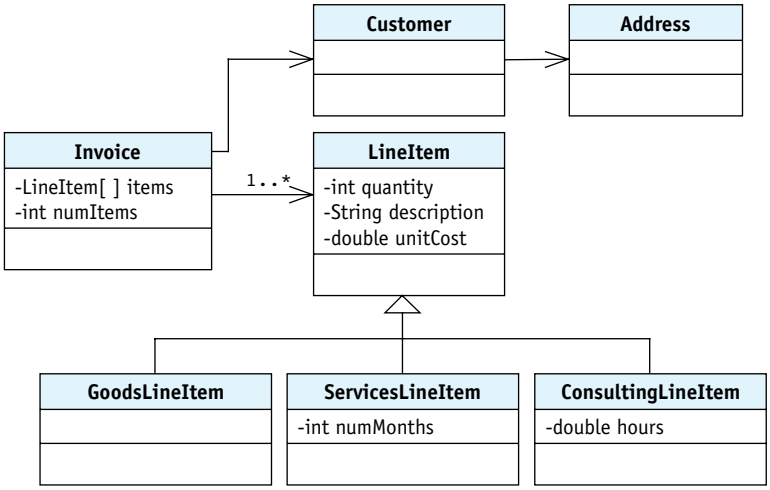
Perhaps the problem with these sentences is the names we've chosen. "Service" and "consulting" refer to what the company provided to the customer. In programming the invoicing system, we are really concerned with what goes on the invoice to represent the goods and the consulting. That is, we're most concerned with the line items. The sample invoice shown in Figure 12-9 has five line items. The first line item is for three computers, the second line item is for an office suite, the third is for service contracts, and the last two line items are for consulting.

The three "kinds of items" the specification refers to are three kinds of line items. If we name them `GoodsLineItem`, `ServicesLineItem`, and `ConsultingLineItem`, then an is-a statement like "A `GoodsLineItem` is a kind of `LineItem`" makes sense. We can conclude that inheritance is appropriate.

These relationships are shown in Figure 12-11. Observe the striking resemblance to the common pattern for polymorphism shown in Figure 12-4.

(figure 12-11)

Initial class diagram for the invoicing system



Assigning Attributes

Some of the nouns and noun phrases will correspond to attributes in these classes. An initial assignment is also shown in Figure 12-11. The `Customer` and `Address` classes are not relevant to the main topics of this chapter and are omitted from the rest of the discussion.

We know from the specification's second paragraph that each line item shows a quantity, description, unit cost, and amount. These seem like good attributes to add to the `LineItem` class. Before we do that, however, we should check two things. First, is it better to compute the value or store it in an attribute? The amount seems like a value that is better computed by a method than stored in an attribute, especially given the extensive explanations about how to calculate it from other values.

KEY IDEA

If an attribute is in a superclass, it should be applicable to all the subclasses.

Second, before placing these attributes in the `LineItem` class we should ask whether they apply to all of `LineItem`'s subclasses. A quick glance at the sample invoice shows that each kind of line item shows all the values. Therefore we conclude that quantity, description, and unit cost can go into `LineItem`.

KEY IDEA

Some attributes do not *belong in the superclass.*

The time spent consulting and the number of months a service is provided are obviously unique to `ConsultingLineItem` and `ServicesLineItem`, respectively.

The remaining attributes all seem to be variations of attributes we have already discussed.

12.2.2 Step 2: Identifying Services

Step 2 of the object-oriented design methodology in Figure 12-8 is to identify potential services by considering the verbs in the specification. The verbs, with slight transformations to show context, are shown in Figure 12-12.

print an invoice
~~request payment for items~~
~~provide items to a customer~~
show customer name, address, total amount billed
add a line item
show line item info (quantity, description, unit cost, total amount)
calculate the amount charged for goods
calculate the amount charged for services
calculate the amount charged for consulting

(figure 12-12)

Verbs from the problem's specification

As with nouns, some of the verb phrases may not belong. For example, "request payment" describes the purpose of the invoice and "provide items to a customer" is something the company does. Neither are things that this computer system should do. Both are crossed off the list.

Assigning Methods to Classes

Printing an invoice is an activity of the `Invoice` class and is assigned there, as is adding a line item. Showing the quantity, description, unit cost, and amount are associated with all line item objects, so we will assign these to the `LineItem` class. They are most likely to be used by the `print` method to get the associated values, so we'll name them `getX` rather than `showX` (where *X* is replaced with a name).

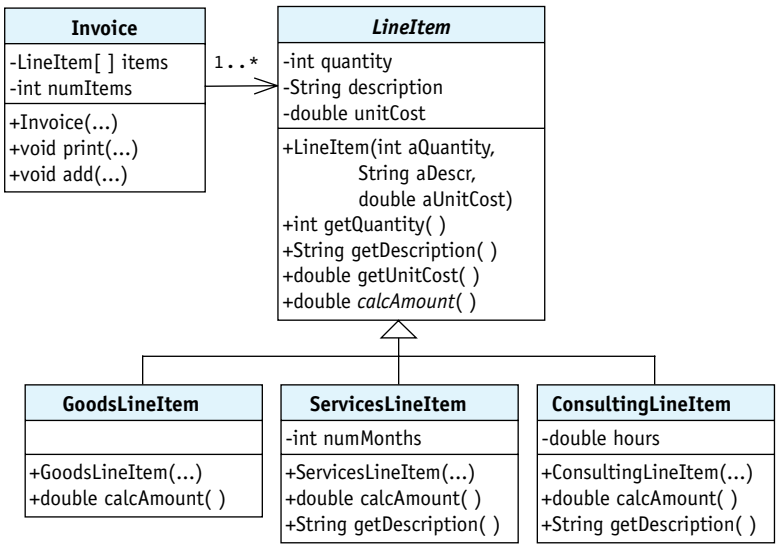
Each line item must calculate the amount to charge for the goods, services, or consulting it represents. On the other hand, we also know that these values are all calculated differently, strongly suggesting that `calcAmount` should be an abstract method in `LineItem`. This allows it to be called polymorphically, but defers the decision of how to calculate the amount to the appropriate subclasses.

The sample invoice shows that the description is displayed differently for each kind of line item. A `ConsultingLineItem` displays the number of hours and a `ServicesLineItem` displays the number of months. This seems similar to `calcAmount`, suggesting another abstract method. However, we also need an accessor method in `LineItem` for `description`, suggesting an accessor method that is overridden as needed in the subclasses.

Figure 12-13 shows the class diagram with these assignments made.

(figure 12-13)

Methods assigned to classes



Implementing Methods

Relevant portions of the `LineItem` class are shown in Listing 12-4. There is nothing unusual about it except for the abstract method to calculate the line item's amount (line 23) and the resulting `abstract` keyword applied to the class (line 4).

FIND THE CODE

ch12/invoice/

Listing 12-4: *The* `LineItem` *class with an abstract method*

```
1  /** A line item is one kind of thing provided by the company for the customer.
2  *
3  *  @author Byron Weber Becker */
4  public abstract class LineItem extends Object
5  {
6    private int quantity;
7    private String description;
8    private double unitCost;
9
10   /** Construct a new line item.
11   *  @param aQuantity      The number of things provided to the customer.
12   *  @param aDescr         A description of the things provided.
13   *  @param aunitCost      The cost of each of the things. */
14   public LineItem(int aQuantity, String aDescr,
15                   double aUnitCost)
16   { super();
17     this.quantity = aQuantity;
18     this.description = aDescr;
```

Listing 12-4: *The* `LineItem` *class with an abstract method* (continued)

```
19      this.unitCost = aUnitCost;
20    }
21
22    /** Calculate the total amount owing due to this line item. */
23    public abstract double calcAmount();
24
25    // Accessor methods omitted.
26  }
```

The interesting methods in `ServicesLineItem` are implemented as shown in Listing 12-5. Invoicing for services requires knowing the number of months the service is provided, resulting in the instance variable `numMonths` in line 6. A value to initialize `numMonths` is passed as an argument to the constructor, also with values to initialize the superclass. It is common for a subclass' constructor to have more parameters than the superclass. It uses some of them to initialize its own instance variables, and passes the rest of them to the superclass.

This class provides a body for `calcAmount` (lines 20–23). Because the quantity and unit cost of the service contracts are stored in `LineItem`, accessor methods are used to get their values.

The `getDescription` method is overridden to add the number of months to the description.

Listing 12-5: *Implementing the interesting methods in the* `ServicesLineItem` *class*

FIND THE CODE

ch12/invoice/

```
1  /** Invoice the customer for 1 or more identical service contracts.
2  *
3  *  @author Byron Weber Becker */
4  public class ServicesLineItem extends LineItem
5  {
6    private int numMonths;
7
8    /** Construct a new line item for services provided.
9    *  @param aQuantity      The number of service contracts provided to the customer.
10   *  @param aDescr         A description of the services provided.
11   *  @param aMthlyCost     The monthly cost of each service contract.
12   *  @param aNumMonths     The number of months the service contract lasts. */
13   public ServicesLineItem(int aQuantity, String aDescr,
14                           double aMthlyCost, int aNumMonths)
15   { super(aQuantity, aDescr, aMthlyCost);
```

Listing 12-5: *Implementing the interesting methods in the* `ServicesLineItem` *class* (continued)

```
16      this.numMonths = aNumMonths;
17    }
18
19    /** Calculate the total amount owing due to this line item. */
20    public double calcAmount()
21    { return this.getQuantity() * this.getUnitCost()
22                                * this.numMonths;
23    }
24
25    /** Get the description of the services represented by this line item. */
26    public String getDescription()
27    { return super.getDescription() +
28                  " (" + this.numMonths + " months)";
29    }
30  }
```

12.2.3 Step 3: Solving the Problem

The last step in the object-oriented design methodology shown in Figure 12-8 is to "apply the services from Step 2 to the objects from Step 1 in a way that solves the problem." We won't solve the entire problem here. We will focus on the `print` method in `Invoice` to show how it uses polymorphism and the inheritance hierarchy we've built. We'll also briefly discuss how to read invoices from a file.

Printing Invoices

The following pseudocode for `print` follows directly from the sample invoice shown in Figure 12-9.

PATTERN
Polymorphic Call

```
print the company's address
print the customer's address
print column headers
totalAmountBilled = 0
for each line item
{  print the quantity, description, unit cost and amount
   totalAmountBilled = totalAmountBilled + amount
}
print totalAmountBilled
```

This code is polymorphic because it does not need to know or care what kind of line item object is in the array of line items. Thanks to polymorphism, the `print` method

can simply call the `calcAmount` and `getDescription` methods and they will return a value appropriate to their actual type.

However, we should recall the lessons learned in Chapter 11. The code inside the loop separates the processing (printing a line item) from the data (information stored in the line item). A better design would keep the data and processing together by placing a `print` method in the `LineItem` class. The `print` method in the `Invoice` class calls `LineItem`'s `print` polymorphically, as shown in Listing 12-6.

FIND THE CODE

ch12/invoice/

Listing 12-6: *The simplified* `print` *method in the* `Invoice` *class*

```
1  public class Invoice extends Object
2  {
3    private LineItem[] items = new LineItem[1];
4    private int numItems = 0;
5    // Constructors, methods, and some instance variables omitted.
6
7    public void print(PrintWriter out)
8    { this.printCompanyAddress(out);
9      this.printCustomerAddress(out);
10     this.printColumnHeaders(out);
11
12     double totalAmountBilled = 0.0;
13     for (int i = 0; i < this.numItems; i++)
14     { LineItem item = this.items[i];
15       item.print(out);                    // polymorphism
16       double amt = item.calcAmount();     // polymorphism
17       totalAmountBilled = totalAmountBilled + amt;
18     }
19
20     this.printTotal(out, totalAmountBilled);
21   }
22 }
```

The `print` method itself is shown in Listing 12-7.

FIND THE CODE

ch12/invoice/

Listing 12-7: *A method to print one* `LineItem`

```
1  public abstract class LineItem extends Object
2  {
3    private static final NumberFormat money =
4            NumberFormat.getCurrencyInstance();
5    // Some instance variables, constructors, and methods omitted.
6
```

Listing 12-7: *A method to print one* `LineItem` (continued)

```
7     /** Print this line item to the specified file. */
8     public void print(PrintWriter out)
9     { out.printf("%3d %-50s%10s%10s%n",
10            this.getQuantity(),
11            this.getDescription(),
12            this.money.format(this.unitCost),
13            this.money.format(this.calcAmount()));
14    }
15 }
```

KEY IDEA

Polymorphism can also occur when an overridden method is called from a superclass.

Polymorphism is at work in this example in two ways. The first is calling `print` polymorphically from the `Invoice` class. The second is that each use of the keyword `this` inside `LineItem`'s `print` method refers to one of the three concrete classes. Therefore, `this.getDescription()` will search for the method `getDescription` beginning with the concrete class, one of `GoodsLineItem`, `ServicesLineItem`, or `ConsultingLineItem`. If `getDescription` was overridden, the more specialized version will be called. Furthermore, when `calcAmount` is called in line 13, the version in this line item's concrete class will be called. This is polymorphism because the client, `LineItem`, doesn't need to know or care what kind of line item it is. Because it is calling methods that may be overridden, this method has a lot in common with the Template Method pattern studied in Section 3.5.3.

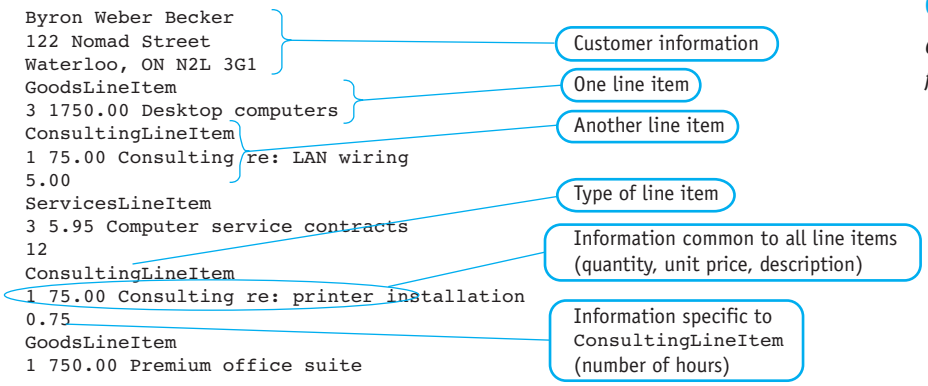
Reading an Invoice from a File

Reading an invoice from a file is trickier than the examples covered in Chapter 9 because of polymorphism. The file must contain all the information needed to reconstruct the different kinds of line items. This has two implications. First, the file must indicate which of the various subclasses of `LineItem` to construct; second, the file must store more data for some line items than for others.

KEY IDEA

Include data in the file that says what kind of subclass to construct.

One possible file format is shown in the example in Figure 12-14. It contains customer information followed by the line items. Each line item uses two or more lines. The first line in each group is a string indicating which class to construct. The remaining lines in the group contain the data used to initialize the objects.



(figure 12-14)

One possible file format for storing line items

An `Invoice` constructor that reads this file is shown in Listing 12-8. It repeatedly reads a line identifying the type of line item required (line 13). The cascading-`if` statement in lines 14–22 calls the appropriate constructor based on the name that was read. By the time control returns to line 12, all of the data for that line item has been read, and the program is ready to read the name of the next subclass.

Listing 12-8: *A constructor for the* `Invoice` *class*

```
1  public class Invoice extends Object
2  {
3    private LineItem[] items = new LineItem[1];
4    private int numItems = 0;
5    // Some constructors, methods, and instance variables omitted.
6
7    /** Read an invoice from a file. */
8    public Invoice(Scanner in)
9    { this.customer = new Customer(in);
10
11     // Read and construct the line items, putting them in the array.
12     while (in.hasNextLine())
13     { String subclass = in.nextLine();
14       if (subclass.equals("GoodsLineItem"))
15       { this.addLineItem(new GoodsLineItem(in));
16       } else if (subclass.equals("ServicesLineItem"))
17       { this.addLineItem(new ServicesLineItem(in));
18       } else if (subclass.equals("ConsultingLineItem"))
19       { this.addLineItem(new ConsultingLineItem(in));
20       } else
21       { throw new Error("Unknown subclass: " + subclass + ".");
22       }
23     }
24   }
```

FIND THE CODE

ch12/invoice/

Listing 12-8: *A constructor for the* `Invoice` *class* (continued)

```
25
26    /** Add one line item to items array. Enlarge the array, if necessary. */
27    public void addLineItem(LineItem item)
28    { // Remainder of method omitted.
29    }
30 }
```

KEY IDEA

Each class reads the data it needs to initialize itself.

The remaining task is to write the constructors needed to read a line item. A total of four are required: one for `LineItem` and one for each of the subclasses. The `LineItem` constructor will be called using `super` in each of the subclass constructors. After it has read the information it requires, reading will resume in the subclass constructor. It reads any remaining information to initialize its own instance variables. Listing 12-9 shows the relevant code for `LineItem`, and Listing 12-10 shows the relevant code for `ConsultingLineItem`.

FIND THE CODE

ch12/invoice/

Listing 12-9: *A constructor to read information for one* `LineItem` *object from a file*

```
1  public abstract class LineItem extends Object
2  { private int quantity;
3    private double unitCost;
4    private String description;
5
6    public LineItem(Scanner in)
7    { super();                    // Read only the data stored in this class.
8      this.quantity = in.nextInt();
9      this.unitCost = in.nextDouble();
10     this.description = in.nextLine();
11   }
12
13   // Remainder of class omitted.
14 }
```

Listing 12-10: *A constructor showing how to use a constructor in the superclass*

FIND THE CODE

ch12/invoice/

```
1  public class ConsultingLineItem extends LineItem
2  { private double hours;
3
4    public ConsultingLineItem(Scanner in)
5    { super(in);       // Superclass reads what it needs from the file, leaving the
6                       // file cursor just before the number of consulting hours.
7      this.hours = in.nextDouble();
8      in.nextLine();
9    }
10
11   // Remainder of class omitted.
12 }
```

Using a Factory Method

The approach shown in the preceding listings works. Its disadvantage is the complexity in the `Invoice` constructor that has little to do with invoices and much to do with line items.

A better approach is to move the complexity of determining which subclass to construct and the actual construction into a static method named `read` in the `LineItem` class. That method can determine which kind of line item is next in the file, construct one, and return it. `read` must be a method and not a constructor because a method can return a subclass of `LineItem`—something a constructor can't do.

KEY IDEA

A static method can return the required subclass of `LineItem`.

The `read` method, shown in Listing 12-11, is very similar to lines 13–22 in Listing 12-8. It must be `static` so that it can be called without an instance of an object.

Listing 12-11: *A factory method*

FIND THE CODE

ch12/invoice/

```
1  public abstract class LineItem extends Object
2  { // Instance variables, constructors, and most methods omitted.
3
4    public static LineItem read(Scanner in)
5    { String subclass = in.nextLine();
6      if (subclass.equals("GoodsLineItem"))
7      { return new GoodsLineItem(in);
8      } else if (subclass.equals("ServicesLineItem"))
9      { return new ServicesLineItem(in);
10     } else if (subclass.equals("ConsultingLineItem"))
```

PATTERN

Factory Method

Listing 12-11: *A factory method* (continued)

```
11      { return new ConsultingLineItem(in);
12      } else
13      { throw new Error("Unknown subclass: " + subclass + ".");
14      }
15    }
16  }
```

This simplifies the constructor in `Invoice`, as shown in Listing 12-12.

Listing 12-12: *Reading line items using a factory method*

```
1  public Invoice(Scanner in)
2  { this.customer = new Customer(in);
3
4      // Read and construct the line items, putting them in the array.
5      while (in.hasNextLine())
6    { this.addLineItem(LineItem.read(in));
7    }
8  }
```

12.3 Polymorphism without Arrays

So far most of our examples of polymorphism have used an array. For example, consider the following statement:

```
double amt = this.items[i].calcAmount();
```

It calls `calcAmount` polymorphically because the array might hold a `GoodsLineItem` or a `ConsultingLineItem`—and this code fragment doesn't need to know or care.

Arrays, however, are *not* a requirement for using polymorphism. In Listing 12-7 we called `this.getDescription()` and the correct subclass of `LineItem` returned the answer.

In fact, the potential for polymorphism exists any time you have a reference to an object. What are some other examples?

KEY IDEA

Polymorphism is a possibility any time you have a reference to an object.

A method may return a reference that is used polymorphically. For example, `Invoice` might have a method to return the most expensive line item, for printing in a report. The report's method could use it this way:

```
LineItem expensive = anInvoice.getMostExpensiveLineItem();
double cost = expensive.calcAmount();
```

The call to `calcAmount` is polymorphic because this code does not need to know what kind of `LineItem` it's dealing with. In fact, this code could be written without using the variable `expensive`:

```
double cost =
  anInvoice.getMostExpensiveLineItem().calcAmount();
```

A reference can also be passed to a parameter, allowing for polymorphism within a method. For example, suppose we had a method with the signature `void gatherStatistics(LineItem item)`. Inside the method, it can call any of the methods declared by `LineItem` without knowing whether it's really a `GoodsLineItem`, a `ServicesLineItem`, or a `ConsultingLineItem`.

An instance variable can also hold an object reference that is used polymorphically.

12.4 Overriding Methods in `Object`

With a new understanding of inheritance and polymorphism, we are now in a better position to understand some of the methods in the class `Object`. There are three that we need to discuss: `toString`, `equals`, and `clone`.

12.4.1 `toString`

Overriding `toString` was discussed in Section 7.3.3. There isn't much to add here except to note that we now know in more detail how Java chooses which `toString` method to execute—and that when `toString` is called, thanks to polymorphism, the caller doesn't need to know or care which subclass of `Object` calculates the answer.

12.4.2 equals

LOOKING BACK

The `DateTime` *class in the* `becker` *library includes the notion of time. We're ignoring that here.*

Section 8.2.4 discussed comparing objects for equivalence. The example was to check whether two dates "mean" the same thing. We discovered that comparing them with == was not the right thing to do. To check for equivalence, a method is required. At that point we wrote the following method:

```
 1  public class DateTime extends Object
 2  { private int year;
 3    private int month;
 4    private int day;
 5
 6    // Other methods omitted.
 7
 8    /** Return true if this date represents the same date as other. */
 9    public boolean isEquivalent(DateTime other)
10    { return other != null && this.year == other.year &&
11             this.month == other.month && this.day == other.day;
12    }
13  }
```

This method is fine except that the designers of Java provide a method in the `Object` class for this purpose: `boolean equals(Object other)`. Their intent is that we override `equals` with the correct implementation for classes we write.

We can't simply change "`isEquivalent`" to "`equals`" in the preceding code because that would produce two different method signatures—the `equals` method in the `Object` class takes an `Object` as its argument whereas the `equals` method in `DateTime` takes a `DateTime` object as its argument. This provides overloading but not overriding, and makes a difference as well. Suppose we have two objects:

```
Object d1 = new DateTime(2008, 1, 1);
DateTime d2 = new DateTime(2008, 1, 1);
```

`d2.equals(d1)` calls the method with the signature `equals(Object other)` (returning `false`) while `d2.equals(d2)` calls the method with the signature `equals(DateTime other)` (returning `true`).

KEY IDEA

Use the right signature to override `equals`*. Overloading isn't good enough.*

To override `equals` correctly, we must use the same signature as defined in `Object`: `public boolean equals(Object other)`.

In the `isEquivalent` method, we know that the object passed via the parameter is a `DateTime` object. With `equals`, any object at all may be passed. We first need to verify that `other` is an instance of the right type, `DateTime`. Fortunately, Java provides a Boolean operator for that purpose. If `x` is a reference variable and `T` is the name of a class or interface, then `x instanceof T` returns `true` if `x` is a non-`null` reference

that can call all of the methods specified in `T`. The type of `x` might be `T`, a subclass of `T`, or a class that implements the interface `T`.

We can use `instanceof` as a key part of our `equals` method, as follows:

KEY IDEA

`instanceof` is used to verify the type of an object.

```
if (!(other instanceof DateTime))
{  // other isn't a DateTime object, so it can't possibly be equal to this DateTime object.
   return false;
}
```

However, if `other` is an instance of `DateTime`, we need to access its fields or methods to compare the dates. Because `other` is declared as an `Object`, we can't just call `other.getYear()`. We need to first assign it to a `DateTime` reference, but Java will not allow us to simply perform the following assignment because it can't verify at compile time that `other` will refer to a `DateTime` object.

```
DateTime dt = other;      // will not compile
```

We can tell the compiler to make an exception with a **cast**. A cast is our assurance to the compiler that we believe `other` will, in fact, refer to a `DateTime` object when the code executes. The compiler doesn't completely trust us, however. It will verify at runtime that `other` can substitute for an object of the specified type. If it cannot, a `ClassCastException` will be thrown.

The syntax for casting an object is like that for casting a primitive type, as in the following:

```
DateTime dt = (DateTime)other;
```

KEY IDEA

Casting an object reference changes the program's interpretation of the object.

The meaning, however, is different. When casting a primitive type, the value is actually changed. For example, `int i = (int)3.99999` assigns `i` the value `3`. When an object reference is cast, the type of the object doesn't change; it's the program's interpretation of the object that changes. Instead of interpreting it as an instance of `Object`, the program now interprets it as an instance of what it really is, `DateTime`.

After casting `other` to `dt`, we can perform the comparisons as in `isEquivalent`. Recall that this code is inside the `DateTime` class, so we can access instance variables via `dt` as well as via `this`:

```
return this.year == dt.year && this.month == dt.month
        && this.day == dt.day;
```

Lastly, an object is compared to itself surprisingly often. This test can be performed very efficiently with `==` and is often included before any of the other tests discussed here.

The complete `equals` method is as follows:

Equals

```
1  public boolean equals(Object other)
2  { if (this == other)
3      return true;          // other is exactly the same object as this.
4
5    if (!(other instanceof DateTime))
6      return false;         // other is not an instance of DateTime (or a subclass).
7
8    // Compare the relevant fields for equality.
9    DateTime dt = (DateTime)other;
10   return this.year == dt.year && this.month == dt.month
11         && this.day == dt.day;
12 }
```

KEY IDEA

Many classes should not override equals.

When should `equals` be overridden? Classes that represent a value such as `Integer`, `DateTime`, or `Color` should have their own `equals` method. Classes where an object is only equal to itself should not. Examples include `Student` (two students may have the same name, but they are not equal to each other) or `BankAccount` (my account shouldn't be "equal" to your account, even if the balances happen to be the same).

Finally, a warning. Whenever `equals` is overridden, a method named `hashCode` should also be overridden, but that's beyond the scope of this book. If you use your class with a class from the Java libraries that includes the word "Hash", watch out! You may get strange results if you override `equals` but not `hashCode`.

12.4.3 `clone` (advanced)

Sometimes an exact copy of an object is required. Suppose, for example, that a back order in our invoicing system begins by requesting a duplicate of a line item. The Java system provides a convention for providing this service based on the `clone` method that all classes inherit from the `Object` class.

The `clone` method has the following goals:

- `x.clone() != x` (the object returned by cloning `x` is not the original object).
- `x.clone().equals(x)` (the cloned object is equal to the original object).
- `x.clone().getClass() == x.getClass()` (the clone and the original have the same run-time class; one is not a subclass of the other).

Using Clone

Suppose that someone has already implemented `clone` for the `LineItem` class. We could then use it to create a duplicate line item object like this:

```
LineItem duplicate = (LineItem)aLineItem.clone();
```

The cast to a `LineItem` is required because the `clone` method is declared to return an `Object`.

Polymorphism comes into play here because `clone` may be overridden in the subclasses of `LineItem`, but we don't need to know or care. The correct method will be called for the actual run-time type of `aLineItem`.

Implementing Clone

The `clone` method in the `object` class implements the following pseudocode:

```
newObject = a new object with the same run-time class as this
for(each instance variable in this)
{ copy the variable's value to newObject
}
return newObject;
```

The `clone` method in `Object` returns an object with the same run-time type as the original object and the same values for all its instance variables.

It may sound like the existing `clone` method is all that's required. Unfortunately, that's not the case. It's dangerous to call `clone` unless issues have been thought about carefully for subclasses (more on these issues will follow). To help ensure that `clone` cannot be called without thinking these issues through, Java's designers have done two things. First, `clone` is protected, meaning it can only be called by a subclass. Therefore, the only way to effectively use `clone` is to override the method and declare it public.

Second, the `clone` method implemented in `Object` checks to make sure that the class implements the `Cloneable` interface. If it doesn't, `clone` throws the `CloneNotSupportedException`. Either that exception must be caught or your `clone` method must declare that it also throws the exception. It's worth noting that this is an unusual use of an interface; it affects the behavior of an existing method rather than guaranteeing the presence of methods. Many programmers believe that this design is a serious mistake. Nevertheless, `clone` is used widely enough that it's worth understanding how it works.

Taking these things into account, an appropriate implementation of `clone` in the `LineItem` class would be as shown in Listing 12-13. Implemented this way, it will also work for the subclasses of `LineItem` discussed earlier in the chapter.

FIND THE CODE

ch12/invoice/

Listing 12-13: *An implementation of* `clone` *in the* `LineItem` *class*

```
1  public abstract class LineItem extends Object
2                        implements Cloneable
3  {
4     // Instance variables, constructors, and methods omitted.
5
6     /** Make a duplicate copy of this object. */
7     public Object clone()
8     { try
9       { return super.clone();
10      } catch (CloneNotSupportedException e)
11      { // CloneNotSupportedException should never be thrown because we have
12        // implemented Cloneable. Error is an unchecked exception.
13        throw new Error("Should never happen.");
14      }
15    }
16 }
```

Dangers: Shallow Copies vs. Deep Copies

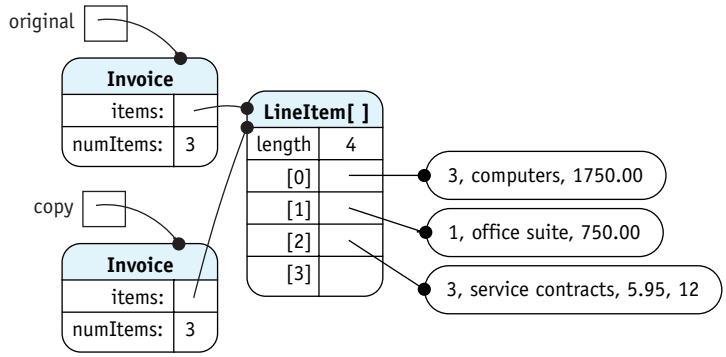
It seems like the `clone` method in the `Object` class does everything required. Why is it so thoroughly protected? The `clone` method in `Object` simply copies the value in each instance variable from the original object to the new object. For primitive types like integers, characters, and immutable classes like `String`, this works very well. For reference types, it often does not.

LOOKING BACK

The value in a reference variable is the address of an object, not the object itself. See Section 8.2.1.

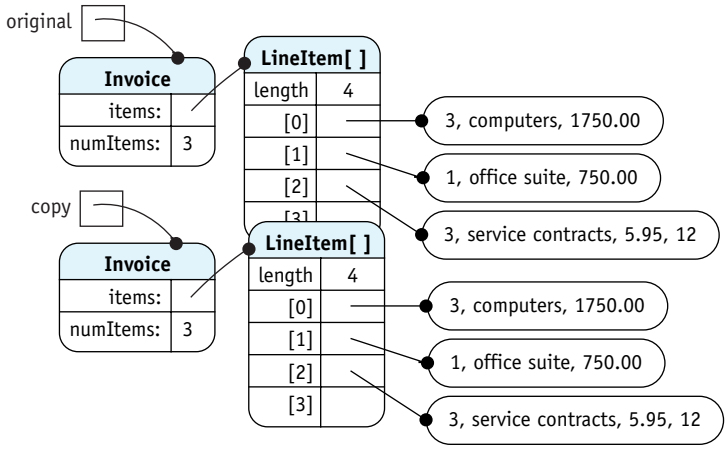
Consider cloning an `Invoice` object. The `items` instance variable refers to an array. The value it stores is a reference to the array, not the array itself. If we call `clone` to clone the invoice, it will copy this array reference but it won't make a copy of the array itself. Both invoices then refer to the same array of line items. If a line item is deleted from the copy of the invoice, it would also be deleted from the original invoice. However, the original's `numItems` variable would *not* be updated, probably leading to nasty results.

Figure 12-15 shows what it is known as a **shallow copy**. That's where only the values in the instance variables are copied from one object to the other. Cloning an invoice should make a **deep copy**. A deep copy also clones objects that the object references. The result of a deep copy is shown in Figure 12-16.



(figure 12-15)

Shallow copy of an `Invoice` *object*



(figure 12-16)

Deep copy of an `Invoice` *object*

For a deep copy, we need to create a new array and clone each element in the old array. The `clone` method in Listing 12-14 shows how.

Listing 12-14: *A clone method that does a deep copy of* `Invoice`

```
1  public class Invoice extends Object implements Cloneable
2  { private LineItem[] items = new LineItem[5];
3    private int numItems = 0;
4
5    // Constructors and methods omitted.
6
7    /** Make a copy of this invoice. */
8    public Object clone()
9    { try
10     { Invoice copy = (Invoice)super.clone();
```

FIND THE CODE

ch12/invoice/

Listing 12-14: *A clone method that does a deep copy of* `Invoice` (continued)

```
11          // Do a deep copy of the array of line items.
12          copy.items = new LineItem[this.numItems];
13          for (int i = 0; i < this.numItems; i++)
14          { copy.items[i] = (LineItem)this.items[i].clone();
15          }
16          return copy;
17        } catch (CloneNotSupportedException e)
18        { throw new Error("Should never happen.");
19        }
20      }
21  }
```

KEY IDEA

Immutable objects can't change and thus don't need to be cloned.

Sometimes a deep copy is not needed, even though references are being used. If the reference is to an immutable object such as `String`, a shallow copy is sufficient. Immutable objects can't change after they are constructed, so there is no danger in having the clone and the original object share the same strings—or any other immutable object.

12.5 Increasing Flexibility with Interfaces

Using interfaces appropriately can allow for more flexible use of the code we write. Flexible code can be used in more situations, often enabling us to avoid writing new code. As an example, we'll explore how the sorting method developed in Section 10.1.8 can be refactored for use in many situations. To make the example concrete, we'll sort instances of `LineItem` in a variety of ways. In our first example, we will consider how to sort `LineItem` by description.

The original sorting method is reproduced in Listing 12-15. The statements shown in bold must change to sort `LineItem` objects. The required changes fall into three categories:

- The documentation, which is very specific to the original project
- The references to a specific array to sort
- The condition used to sort the array

Listing 12-15: *The sorting algorithm from the Big Brother/Big Sister project. Required changes to sort line items are shown in bold*

```
1  public class BBBS extends Object
2  { ... persons ...          // an array of Person objects
3
4    /** Sort the persons array in increasing order by age. */
```

Listing 12-15: *The sorting algorithm from the Big Brother/Big Sister project. Required changes to sort line items are shown in bold.* (continued)

```
5    public void sortByAge()
6    { for (int firstUnsorted=0;
7           firstUnsorted < this.persons.length - 1;
8           firstUnsorted++)
9      { // Find the index of the youngest unsorted person.
10       int extremeIndex = firstUnsorted;
11       for (int i = firstUnsorted + 1;
12             i < this.persons.length; i++)
13       { if (this.persons[i].getAge() <
14             this.persons[extremeIndex].getAge())
15         { extremeIndex = i;
16         }
17       }
18
19       // Swap the youngest unsorted person with the person at firstUnsorted.
20       Person temp = this.persons[extremeIndex];
21       this.persons[extremeIndex] =
22             this.persons[firstUnsorted];
23       this.persons[firstUnsorted] = temp;
24     }
25   }
26 }
```

Of the three categories of change identified earlier, the first two are easy. Generalizing the documentation is trivial, and the problems with the names can be handled with appropriate parameters. Once we use parameters, all reliance on instance variables is removed, and the `sort` method can be made a class (`static`) method in a utilities class. The first set of changes is shown in Listing 12-16. The only part left is to figure out how to replace the pseudocode in line 10, which will influence the type of array passed as an argument in line 4 and the type of temporary variable in line 16.

Listing 12-16: *Making* `sort` *more reusable with parameters*

```
1  public class Utilities extends Object
2  {
3    /** Sort an array of objects. */
4    public static void sort(????[] a)
5    { for (int firstUnsorted = 0; firstUnsorted < a.length-1;
6           firstUnsorted++)
```

Listing 12-16: *Making* `sort` *more reusable with parameters* (continued)

```
7       { // Find the index of extreme ("smallest") unsorted element.
8         int extremeIndex = firstUnsorted;
9         for (int i = firstUnsorted + 1; i < a.length; i++)
10        { if (a[i] is less than a[extremeIndex])
11          { extremeIndex = i;
12          }
13        }
14
15        // Swap the extreme unsorted element with the element at firstUnsorted.
16        ???? temp = a[extremeIndex];
17        a[extremeIndex] = a[firstUnsorted];
18        a[firstUnsorted] = temp;
19      }
20    }
21  }
```

12.5.1 Using an Interface

Line 10 of Listing 12-16 requires comparing two elements in the array to determine which is "less" than the other, or which one should occur first in sorted order.

It would be really nice if the `Object` class had an `isLessThan` method similar to the `equals` method. If it did, we could pass an array of `Objects` in line 4 and replace the pseudocode in line 10 with:

```
if (a[i].isLessThan(a[extremeIndex]))
```

and the `sort` method would be done. It would depend, of course, on subclasses of `Object` overriding `isLessThan` appropriately. Unfortunately, `Object` does not provide such a method.

Another approach is to define `isLessThan` in the `LineItem` class and declare `sort` to take an array of `LineItem` objects as its parameter. This works, but only allows `sort` to sort `LineItems` and subclasses of `LineItem`. It would be preferable to have a solution that is much more general.

KEY IDEA

Implementing an interface gives the class an additional type.

An excellent solution is to use an interface. This allows a class such as `LineItem` to have an extra type—the type of the interface. Java already provides such an interface, `Comparable`. It's included in the package `java.lang`, which is automatically imported into every class. The interface is defined as shown in Listing 12-17.

Listing 12-17: *The* `Comparable` *interface from the Java library*

```
1  public interface Comparable
2  { /** Compare this object with the specified object for order. Return a negative number
3        * if this object is less than the specified object, a positive number if this object is greater,
4        * and 0 if this object is equal to the specified object.
5        * @param o The object to be compared. */
6    public int compareTo(Object o);
7  }
```

To use this interface, we need to make the three changes to the `sort` method shown in Listing 12-16:

- In line 4, declare the array parameter variable using `Comparable`: `public static void sort(Comparable[] a)`.
- Declare the type of the temporary variable used to swap elements in line 16 using `Comparable`.
- Change line 10 to `{ if (a[i].compareTo(a[extremeIndex]) < 0)`.

Finally, in any class that we want to sort with this method, we need to implement `Comparable`. To sort the line items by description, we would change `LineItem` as follows:

```
1  public abstract class LineItem extends Object
2                          implements Comparable
3  { private String description;
4
5    // Other instance variables, constructors, and methods omitted.
6
7    public int compareTo(Object o)
8    { LineItem item = (LineItem)o;
9      return this.description.compareTo(item.description);
10   }
11 }
```

The class declaration in lines 1 and 2 includes the phrase `implements Comparable`. The only method it specifies is declared in lines 7–10. Notice that in line 8, the object is cast to a `LineItem`. This is necessary to gain access to the instance variables required to do the comparison. This cast also works for subclasses of `LineItem` but will fail if `o` is something else, like a `Robot`. In that case, Java will throw a `ClassCastException` to indicate an error. The documentation in the `Comparable` interface says that this is what *should* happen when two objects can't be compared to each other.

KEY IDEA

Interfaces are another way to exploit polymorphism in a program.

What do sorting and interfaces have to do with polymorphism? Thanks to polymorphism, the `sort` method can call the `compareTo` method without knowing or caring which class actually implemented it. The `sort` method doesn't care whether the `compareTo` method is comparing descriptions or unit costs or the total cost of the line item.

Sorting in the Java Library

KEY IDEA

Use Java's sort instead of writing your own.

The Java library includes a sort method very similar to the one we have written except that it is much faster, particularly on large arrays. It's in the `java.util.Arrays` class and has the following signature:

```
public void sort(Object[] a)
```

If you want to sort a partially filled array, you can use a companion method with the following signature:

```
public void sort(Object[] a, int fromIndex, int toIndex)
```

These two methods have arrays of objects as parameters rather than arrays of `Comparable` like our `sort` method. How does that work?

The documentation states that all of the elements must implement `Comparable` and that `compareTo` must not throw an exception for any pair of elements. If these conditions are violated, `sort` will throw a `ClassCastException`. We can make our version of `sort` behave the same way by making two changes to Listing 12-16. First, change the type of the parameter in line 4 from `Comparable[]` to `Object[]`. Second, include a cast inside the loop that calls `compareTo`, as follows:

```
 9       for (int i = firstUnsorted + 1; i < a.length; i++)
10       { if (((Comparable)a[i]).compareTo(a[extremeIndex]) < 0)
11         { extremeIndex = i;
12         }
13       }
```

Mixin Interfaces

A **mixin** is a type that supplements the "primary type" of a class. It provides some behavior that is mixed in with the normal behavior of the primary type. `Comparable` is one such mixin that allows comparing objects and thus sorting them.

Other mixin interfaces include the following:

- `IMove`: The interface we use in Section 12.1.4 to move dancing robots and the banner they carry (a subclass of `JDialog`) is a mixin interface.
- `Runnable`: It is used just for fun in Section 3.5.2 to allow several robots to move simultaneously.

- `Observer`: An interface used when one object wants to "observe" what happens in another. We'll use a variation of this when we write graphical user interfaces in Chapter 13.
- `Paintable`: The class we used in Section 6.1 to ensure that `SimpleBots` could be painted on the screen could just as easily have been a mixin interface.

You may want to define your own interface to use as a mixin when an application needs to process similarly a number of classes that don't have a natural common superclass.

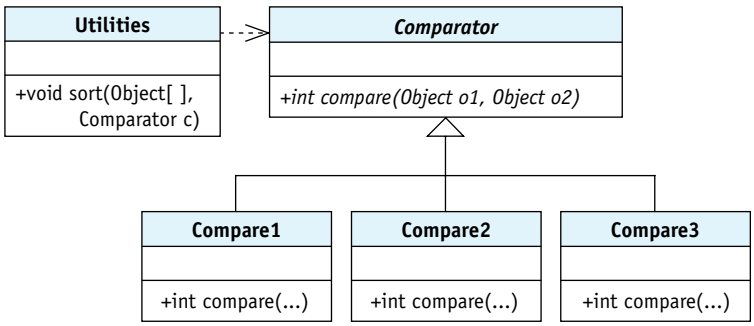
12.5.2 Using the Strategy Pattern

Implementing the `Comparable` interface in `LineItem` is fine if you want to sort the line items in only one way. But suppose you are writing a report program for the marketing department. They want line items from all the invoices gathered into a single report, sorted by total amount. We can't redefine the `compareTo` method just for them, so what do we do?

The **Strategy pattern** uses objects that define a family of interchangeable algorithms. For sorting, we'll use a strategy object that defines the comparison algorithm. When we want a different sort order, we pass a different strategy object (defining a different algorithm) to the `sort` method. This is facilitated with the `Comparator` interface, as shown in Figure 12-17. Notice that the `sort` method in `Utilities` takes an instance of `Comparator` as an argument.

PATTERN

Strategy



(figure 12-17)

Using strategy objects to define the sort order

A Strategy Object Using `Comparator`

The `Comparator` interface is defined in the `java.util` package and is quite similar to `Comparable`. They both define a method that compares two objects and returns an integer whose sign indicates which object is smaller. A key difference is that a `Comparator` is passed both objects as parameters rather than comparing one object to

KEY IDEA

Many different objects can implement `Comparator`*, each comparing objects in its own way.*

itself. A more minor difference is that `Comparator`'s method is named `compare` rather than `compareTo`. The `Comparator` interface is declared as follows:

```
public interface Comparator
{  /** Compare obj1 and obj2 for order. Return a negative number if obj1 is less than
   *  obj2, a positive number if obj1 is greater than obj2, and 0 if they are equal.
   *  @param obj1  One object to be compared.
   *  @param obj2  The other object to be compared. */
   public int compare(Object obj1, Object obj2);
}
```

The following class defines a strategy object that can compare line items when sorting the marketing department's report. Notice that it includes the phrase `implements Comparator` in line 3. Lines 7–9 are formatted differently than we have seen before to save space.

```
 1  /** Compare two line items using the value calculated by calcAmount. */
 2  public class LineItemAmountComparator extends Object
 3                                    implements Comparator
 4  { public int compare(Object obj1, Object obj2)
 5    { double amt1 = ((LineItem)obj1).calcAmount();
 6      double amt2 = ((LineItem)obj2).calcAmount();
 7      if (amt1 < amt2)                { return -1;}
 8      else if (amt1 > amt2)           { return 1; }
 9      else                            { return 0; }
10    }
11  }
```

The `sort` method also needs to take an instance of `Comparator` as a parameter. This is shown in Listing 12-18. The method is just like the previous version of `sort` except for lines 4 and 11. In line 4, there is a new parameter to pass the strategy object implementing the comparison algorithm. In line 10, it's used to compare two line items.

With these changes, we can use `sort` to sort an array of *any* kind of object in any order we want, as long as we can provide a comparison strategy object. That's a lot of flexibility!

KEY IDEA

Use `java.util.Arrays` rather than writing your own sort method.

In practice, however, we would not write our own `sort` routine. We would only write the `Comparator` and use it with the `sort` method in `java.util.Arrays`.

Strategy

Listing 12-18: *A `sort` method that uses a comparator method*

```
1  public class Utilities extends Object
2  {
3     /** Sort a partially-filled array of objects. */
4     public static void sort(Object[] a, Comparator c)
5     { for (int firstUnsorted = 0; firstUnsorted < a.length-1;
```

Listing 12-18: *A sort method that uses a comparator method* (continued)

```
          firstUnsorted++)
      { // Find the index of extreme ("smallest") unsorted element.
        int extremeIndex = firstUnsorted;
        for (int i = firstUnsorted + 1; i < a.length; i++)
        { if (c.compare(a[i], a[extremeIndex]) < 0)
          { extremeIndex = i;
          }
        }

        // Swap the extreme unsorted element with the element at firstUnsorted.
        Object temp = a[extremeIndex];
        a[extremeIndex] = a[firstUnsorted];
        a[firstUnsorted] = temp;
      }
    }
}
```

Sorting with Multiple Keys

Suppose the marketing department wanted a report with all line items sorted first by description, and if the descriptions happen to be the same, then in descending order by total amount of the line item. The description is called the **primary key**. It is the most important determinant of the order. If two objects have different primary keys, then those keys alone are used to determine the order. However, if the primary keys are equal, then the **secondary key** is used to determine the order. In this case, total amount is the secondary key.

Following is a comparator that implements the described ordering:

```
class LineItemDescrTotalComparator extends Object
                              implements Comparator
{ public int compare(Object obj1, Object obj2)
  { LineItem li1 = (LineItem)obj1;
    LineItem li2 = (LineItem)obj2;

    // Compare using the primary key (description).
    int result = li1.getDescription().compareTo(
                           li2.getDescription());
    if (result == 0)       // Primary key is the same; use secondary key.
    { double amt1 = li1.calcAmount();
      double amt2 = li2.calcAmount();
      if (amt1 < amt2)
```

```
14        { result = 1;         // Descending order.
15        } else if (amt1 > amt2)
16        { result = -1;
17        }
18      }
19      return result;
20    }
21  }
```

KEY IDEA

Sort in descending order by reversing the signs of the returned values.

Notice the `if` statement for the secondary key in lines 13–17. Normally we return a negative number when the first argument is less than the second. Here we return positive 1, and –1 when the first argument is larger. Reversing these two values sorts the objects in descending order. Larger amounts are interpreted as "smaller" by this comparator.

Anonymous Classes (advanced)

Small strategy objects such as `Comparator` are so common that Java's designers included a shortcut for defining them quickly and easily. This shortcut is called an **anonymous class**. An anonymous class has the following properties:

- The class doesn't have a name (that's why it's called anonymous).
- It combines declaring a class and instantiating one (and only one) object.
- An anonymous class is defined at the same place the object it defines is needed. This can, if the class is small, improve the understandability of your code.

The following is an example of an anonymous class that sorts line items by description using a sort method from the Java library. To use this code, you must import `java.util.Arrays` and `java.util.Comparator`.

```
1  private void sortLineItems()
2  { // An anonymous class to compare line items by description.
3    Comparator c = new Comparator()
4    {
5      public int compare(Object obj1, Object obj2)
6      { LineItem li1 = (LineItem)obj1;
7        LineItem li2 = (LineItem)obj2;
8
9        return li1.getDescription().compareTo(
10                 li2.getDescription());
11     }
12   };
13
14   Arrays.sort(this.items, c);
15 }
```

The anonymous class appears in lines 3–12. Line 3 looks like any other object instantiation except that the semicolon is missing from the end of the line and `Comparator` is an interface rather than a class. `Comparator` can be replaced by the interface the anonymous class is to implement or the class it is to extend.

The body of the anonymous class appears between the "constructor" and the semicolon terminating the assignment statement. In the previous code, the body appears in lines 4–12.

Because the anonymous class has no name, it can't have a constructor, only methods. It may have instance variables, but they are uncommon and must always be initialized in their declaration because there is no constructor.

An anonymous class can be used to create exactly one object. This one is assigned to the variable `c`. This variable isn't required. In fact, experienced programmers will often replace the variable `c` in line 14 with the code between the equals in line 3 and the semicolon in line 12. However, this practice makes the code more difficult to read.

Applications of Strategy Objects

Strategy objects are widely used for more than sorting. They often have a single method but could have more. Here are a few uses:

- The Java library has a number of static sorting methods in the `java.util.Arrays` class that take a `Comparator` strategy object.
- Strategy objects are used to arrange components in graphical user interfaces. We'll discuss this more in Section 12.6.
- Many games use strategy objects to define different approaches for choosing the next move.
- Several classes within the `becker.robots` package, including `Robot`, include methods—such as `examineThings`—that take a strategy object as an argument. The method returns references to objects the robot may want to "examine." The strategy object determines which objects should be examined.

One particular use for strategy objects is handling objects that change behavior over time. For example, an employee might move from hourly compensation to a salary and perhaps to being compensated by contract over her tenure with a company. Using an inheritance-based approach would require replacing an `HourlyEmployee` object with a `SalariedEmployee` object, for example, as the employee is compensated differently.

Representing this kind of variation with subclasses creates problems as soon as there is more than one kind of variation. Suppose that mode of work (telecommute vs. office) is also represented with subclasses. Now we need `HourlyTelecommutingEmployee`, `SalariedTelecommutingEmployee`, `ContractTelecommutingEmployee`, plus three more for office employees. This quickly becomes unmanageable. Using strategy

objects to represent how employees are paid and how they work is a much better solution. When their compensation method or their mode of work changes, simply replace the strategy object stored in their `Employee` object.

12.5.3 Flexibility in Choosing Implementations

Java interfaces allow several implementations to be used the same way. This can allow you to more easily change our minds later. For example, Java provides a set of classes for storing collections of objects, similar to arrays. Two of these classes are `ArrayList` and `LinkedList`. The two classes have many methods in common: `add`, `remove`, `contains`, and so on. They also implement the same interface, `List`.

Why provide two classes that apparently do exactly the same thing? The answer is that they are implemented differently and have different speed characteristics, as summarized in Table 12-1.

(table 12-1)

Speed characteristics of `ArrayList` and `LinkedList`

Operation	ArrayList	LinkedList
Add an object near the beginning of the list	slow	fast
Add an object near the end of the list	fast	fast
Remove from near the beginning of the list	slow	fast
Remove from near the end of the list	fast	slow
Get an object at a specified position	fast	slow
Set an object at a specified position	fast	slow
Determine if the list contains a specified object	slow	slow
Determine the size of the list	fast	fast

If your application adds and removes objects infrequently but uses `get` a lot, then `ArrayList` looks like a good choice. On the other hand, if `get` is infrequent but there are many additions and deletions, `LinkedList` seems better.

KEY IDEA

Interfaces make it easier to change which class is used.

Sound complicated? Afraid you might make the wrong choice and you'll want to change your mind later? Then use the `List` interface to declare your variables. This can isolate the decision of which class to use to a single point—which constructor to call when the list is first created. If you change your mind, there is only one place to change, and the entire program can take advantage of your new approach.

For example, an inventory program might include a method to remove the items just sold from the items in stock, as sketched in the code fragment shown in Listing 12-19. Note that `List` is used throughout, leaving lots of flexibility to use either `ArrayList`, `LinkedList`, or some other implementation of the `List` interface as the actual class.

Listing 12-19: *A class using interfaces to promote flexibility*

FIND THE CODE

ch12/inventory/

```
1  public class Inventory extends Object
2  { private List<Item> inventory = new ArrayList<Item>();
3    private List<Item> reorder = new LinkedList<Item>();
4    ...
5
6    /**Remove the specified items from the current inventory. Update the list of items
7    *  to reorder.
8    *  @itemsSold  The items that have been sold and need to be removed from inventory. */
9    public void removeInventory(List<Item> itemsSold)
10   { for (Item item : itemsSold)
11     {
12       // Remove the item from the inventory.
13       this.inventory.remove(item);
14
15       // If it's the last one and not already on the reorder list, add it
16       if (!this.inventory.contains(item) &&
17                 !this.reorder.contains(item))
18       { this.reorder.add(item);
19       }
20     }
21   }
22 }
```

By using `List` to declare variables in lines 2, 3, and 9, the programmer has left lots of flexibility to change the actual classes being used. For example, the `ArrayList` in line 2 could be changed to a `LinkedList` with no further changes in the rest of the program.

12.6 GUI: Layout Managers

PATTERN

Strategy

Most graphical user interfaces allow users to interact with many components (buttons, text boxes, sliders, and so on). The issue to be addressed in this section is how Java arranges the components in a panel, both initially and as the user resizes the frame displaying the panel. The task of arranging the components on the panel is called **layout**. Java uses strategy objects called **layout managers** to determine how to arrange the components. By using strategy objects, `JPanel` can display the same set of components in many different arrangements.

12.6.1 The `FlowLayout` Strategy

The default layout strategy for a `JPanel` is an instance of `FlowLayout`. It adds components to the current row until there is no more room. It then starts a new row. The

length of a row is determined by the width of the `JPanel`. Wider panels will have more components on a row.

The left image in Figure 12-18 shows four components organized with a `FlowLayout` strategy. The components are displayed left to right, top to bottom, in the same order they were added. The right image shows how those same components are reorganized when the frame is narrower.

(figure 12-18)

The `FlowLayout` *strategy*

A `FlowLayout` object centers rows by default. It can also be set to align them on either the left or right side of the panel.

Each component has a preferred size, which is respected by `FlowLayout`. As we'll soon see, some layout managers ignore such size information.

12.6.2 The `GridLayout` Strategy

The strategy implemented by a `GridLayout` object is to place all of the components into a grid, as shown in Figure 12-19. Each component is made the same size as all the others, completely ignoring their preferred sizes. The number of rows and columns is set when the strategy object is created.

(figure 12-19)

The `GridLayout` *strategy*

Setting a `JPanel`'s layout strategy is done with its `setLayout` method, as shown in lines 17–18 of Listing 12-20. This listing is already showing the program structure we will adopt for our graphical user interfaces. A group of components is combined by extending `JPanel`. Laying out the components is a distinct task that is delegated to a private helper method called `layoutView`.

Listing 12-21 displays an instance of this panel in a frame.

Listing 12-20: *A* `JPanel` *extended to show a group of buttons, organized with a grid strategy*

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class DemoGridLayout extends JPanel
5  {
6    private JButton one = new JButton("One");
7    private JButton two = new JButton("Two");
8    // Instance variables for the last four buttons are omitted.
9
10   public DemoGridLayout()
11   { super();
12     this.layoutView();
13   }
14
15   private void layoutView()
16   { // Set the layout strategy to a grid with 2 rows and 3 columns.
17     GridLayout strategy = new GridLayout(2, 3);
18     this.setLayout(strategy);
19
20     // Add the components.
21     this.add(this.one);
22     this.add(this.two);
23     // Code to add the last four buttons is omitted.
24   }
25 }
```

FIND THE CODE

ch12/layoutManagers/

PATTERN

Strategy

Listing 12-21: *A* `main` *method that displays a custom* `JPanel` *in a frame*

```
1  import javax.swing.*;
2
3  public class GridLayoutMain
4  {
5    public static void main(String[] args)
6    { JPanel p = new DemoGridLayout();
7
8      JFrame f = new JFrame("GridLayout");
9      f.setContentPane(p);
10     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11     f.pack();              // Base frame size on preferred size of components.
12     f.setVisible(true);
13   }
14 }
```

FIND THE CODE

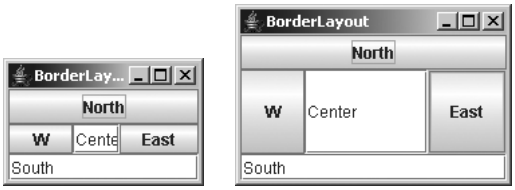
ch12/layoutManagers/

12.6.3 The `BorderLayout` Strategy

The `BorderLayout` strategy lays out up to five objects in a panel, as shown in Figure 12-20. No matter what size the panel is, the north and south areas cover the entire width. Their heights are determined by the preferred heights of the components they hold. The east and west areas expand or contract to occupy the remaining height of the panel. Their widths are determined by the preferred sizes of the components they hold. Finally, the center area expands or contracts to occupy the remaining space.

(figure 12-20)

The `BorderLayout` *strategy*



Areas that do not have a component will not take any space. For example, if the button was left out of the east area in Figure 12-20, the center area would simply expand to fill it.

The layout managers we've seen previously arrange the components according to the order in which they are added to the panel. `BorderLayout` handles positioning with a **constraint**, which is specified when the component is added. The constraint says where the component should be placed.

Listing 12-20 could be modified to use a `BorderLayout` strategy by changing line 17 to:

```
17      BorderLayout strategy = new BorderLayout();
```

and changing the lines that add the components to use the required constraints.

```
21      this.add(this.one, BorderLayout.EAST);
22      this.add(this.two, BorderLayout.NORTH);
```

12.6.4 Other Layout Strategies

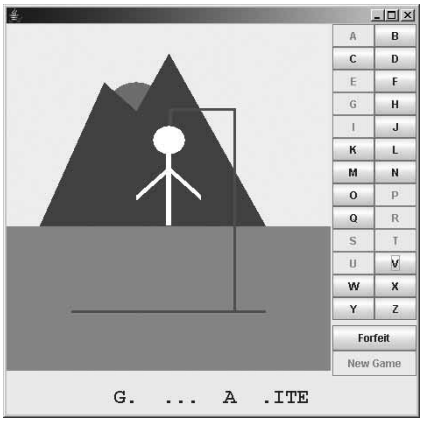
The `BoxLayout` strategy arranges components in a horizontal row or a vertical column. It tries to respect the preferred sizes of components. However, if a component does not have a maximum size, it will grow or shrink to fill available space. Text fields and text areas, for example, do not have a maximum size unless you set one.

Like `GridLayout`, `GridBagLayout` uses a grid. However, its cells can vary in size, and a component can take up more than one cell in the grid. To accomplish all this, it uses a fairly complex constraint, called `GridBagConstraints`.

Another constraint-based layout strategy is `SpringLayout`. It works by specifying how the edges of each component relate to other components or to the edges of the enclosing panel.

12.6.5 Nesting Layout Strategies

A single layout strategy is usually not enough for a complex graphical user interface. Consider Figure 12-21, for example. None of the simpler layout strategies we've covered can handle this by themselves. `GridBagLayout` and `SpringLayout` could do it, but using them would involve a tremendous amount of work in setting all the constraints.

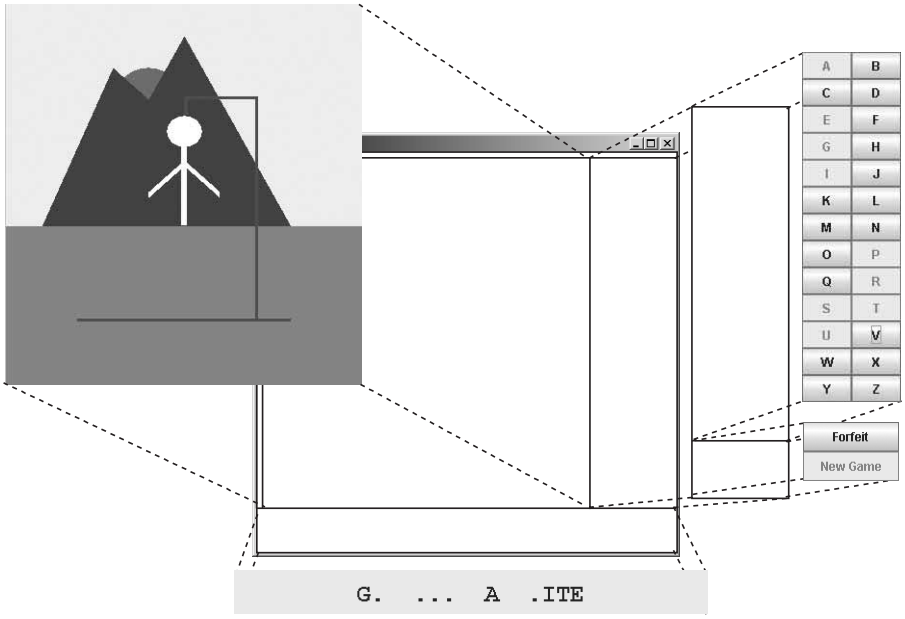


(figure 12-21)
A complex layout task

An excellent solution is based on the fact that `JPanel` is also a component. It can be added to another `JPanel` that is organized by its own layout strategy object. The user interface in Figure 12-21 is organized with four `JPanel` objects, as shown in Figure 12-22.

LOOKING AHEAD

Programming Exercise 12.12 asks you to finish implementing `HangmanView`.



(figure 12-22)
Laying out a complex user interface using nested panels, each with its own layout strategy

The four `JPanel` objects are as follows:

- `controls` is organized by a `GridLayout` and holds the `Forfeit` and `New Game` buttons.
- `letters` is also organized by a `GridLayout` and holds 26 buttons, one for each letter of the alphabet.
- `buttons` is organized by a `BoxLayout` and holds two `JPanel` components, `controls` and `letters`.
- `hangman` is organized by a `BorderLayout`. The center area holds the graphic showing the gallows. The south area holds a `JLabel` displaying the letters guessed so far. The east area holds the `buttons` panel (which holds `letters` and `controls`). The north and west area of the `BorderLayout` are empty and shrink to take no space.

This interface can be implemented with code similar to that shown in Listing 12-22.

FIND THE CODE

ch12/hangman/

Listing 12-22: *Implementing nesting layout managers*

```
1  import becker.xtras.hangman.*;
2  import javax.swing.*;
3  import java.awt.*;
4
5  /** Layout the view for the game of hangman.
6  *
7  *  @author Byron Weber Becker */
8  public class HangmanView extends JPanel
9  { // Constructor omitted.
10
11   /** Layout the view in a JPanel managed by BorderLayout. */
12   private void layoutView()
13   { JPanel hangman = this;        // Use same name as previous discussion
14     hangman.setLayout(new BorderLayout());
15
16     // South
17     JLabel phrase = new JLabel("GO FLY A KITE");
18     hangman.add(phrase, BorderLayout.SOUTH);
19
20     // Center
21     JComponent gallows = new GallowsView(
22                                    new SampleHangman());
23     hangman.add(gallows, BorderLayout.CENTER);
24
25     // East -- letters and controls
26     JPanel buttons = this.buttonsPanel();
```

PATTERN

Strategy

Listing 12-22: *Implementing nesting layout managers* (continued)

```
27      hangman.add(buttons, BorderLayout.EAST);
28    }
29
30    /** Layout and return a subpanel with all the buttons. */
31    private JPanel buttonsPanel()
32    { // A JPanel holding 26 buttons, one for each letter of the alphabet.
33      JPanel letters = new JPanel();
34      letters.setLayout(new GridLayout(13, 2));
35      for (char ch = 'A'; ch <= 'Z'; ch++)
36      { letters.add(new JButton("" + ch));
37      }
38
39      // A JPanel holding the Forfeit and New Game buttons is omitted.
40
41      return letters;
42    }
43  }
```

12.7 Patterns

12.7.1 The Polymorphic Call Pattern

Name: Polymorphic Call

Context: You are writing a program that handles several variations of the same general idea (for example, several kinds of bank accounts). Each kind of thing has similar behaviors, but the details may differ.

Solution: Use a polymorphic method call so that the actual object being used determines which method is called. The most basic form of the pattern is identical to the Command Invocation pattern from Chapter 1 except for how the *«objReference»* is given its value. For example,

```
«varTypeName» «objReference» = «instance of objTypeName»;
...
«objReference».«serviceName»(«parameterList»);
```

where *«objTypeName»* is a subclass of *«varTypeName»* or *«objTypeName»* is a class that implements the interface *«varTypeName»*.

There are many variations. For example, *«objReference»* could be a simple instance variable, an array, a parameter, or a value returned from a method.

Consequences: *«varTypeName»* determines the names of the methods that can be called using *«objReference»*, but *«objTypeName»* determines the code that is actually executed.

Related Pattern: This pattern is a variation of the Command Invocation pattern.

12.7.2 The Strategy Pattern

Name: Strategy

Context: The way an object behaves may change over time or from application to application. Examples include how an employee is compensated as the nature of his or her employment changes, how a game chooses its move as the player adjusts preferences, or how a `JPanel` lays out the components it contains.

Solution: Identify the methods that may need to be executed differently, depending on the strategy. Define these methods in a superclass or an interface. Write several subclasses that implement the behavior required at specific phases in a program's life.

For example, in a game, a player object needs to make its next move depending on the preferences of the user. The `Player` class could be defined as follows, where `MoveStrategy` is either the superclass of several different strategy classes or an interface that is implemented by several strategy classes.

```
public class Player extends ...
{ private MoveStrategy moveStrategy =
            new DefaultMoveStrategy();
  ...
  public void setMoveStrategy(MoveStrategy aStrategy)
  { this.moveStrategy = aStrategy;
  }

  public Move getMove(...)
  { return this.moveStrategy.getMove(...);
  }
}
```

Consequences: The behavior of a class can be easily changed as the program proceeds simply by supplying a different strategy object.

Related Patterns:

- This pattern is a specialization of the Has-a (Composition) pattern.
- The Polymorphic Call pattern is used to call the methods in the strategy object.

12.7.3 The Equals Pattern

Name: Equals

Context: Objects must be compared for equivalency with each other. Comparisons may be done using such library code as `ArrayList` or `HashSet`, and so a standard approach must be used.

Solution: Override the `equals` method in the `Object` class. It is designated to take any instance of `Object` (including subclasses) as its argument, so care must be taken to ensure that the two objects can be compared. The following general template may be used:

```
public class «className» ...
{ private «primitiveType» «primitiveField1»
  ...
  private «primitiveType» «primitiveFieldN»
  private «referenceType» «referenceField1»
  ...
  private «referenceType» «referenceFieldN»

  public boolean equals(Object other)
  { if (this == other)
      return true;

    if (!(other instanceof «className»))
      return false;

    «className» o = («className»)other;
    return
       this.«primitiveField1» == o.«primitiveField1» &&
       ...
       this.«primitiveFieldN» == o.«primitiveFieldN» &&
       this.«referenceField1».equals(o.«referenceField1») &&
       ...
       this.«referenceFieldN».equals(o.«referenceFieldN»);
  }
}
```

where == is used for primitive fields and `equals` is used for object references. It may be that only a subset of the object fields are used to determine equality.

Consequences: The `equals` method can be used to check any object for equivalence with any other object.

Related Pattern: This pattern should be used in place of the Equivalence Test pattern.

12.7.4 The Factory Method Pattern

Name: Factory Method

Context: A specific subclass should be instantiated depending on various factors, such as the information found in a file or values obtained from a user. The logic for deciding which specific subclass to create should be localized in one place in the program.

Solution: Write a method that determines which subclass to instantiate and then returns it. In general,

```
public static «superClassName» «factoryMethodName»(...)
{ «superClassName» instance = null;
  if («testForSubclass1»)
  { instance = new «subclassName1»(...);
  } else if («testForSubclass2»)
  { instance = new «subclassName2»(...);
  } else ...

  return instance;
}
```

Consequences: A specific subclass is chosen to be instantiated and then returned for use.

Related Pattern: None.

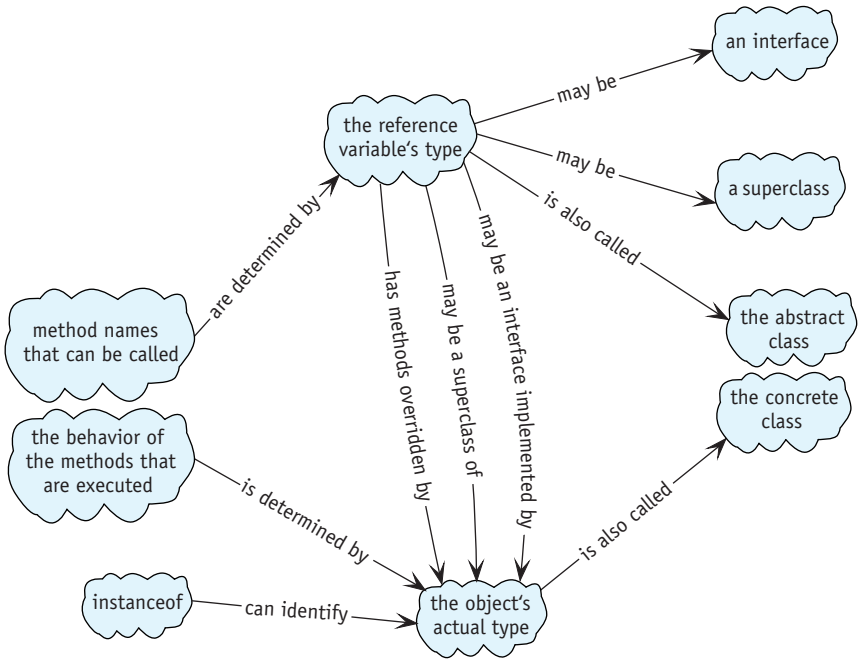
12.8 Summary and Concept Map

Polymorphism is a programming technique in which a variable declared with a superclass or an interface, *X*, is actually assigned an instance of a different class, *Y*. *Y* must be either a subclass of *X* or a class implementing interface *X*.

The program typically calls a method defined by *X* but the behavior is determined by the object's actual class, *Y*. This allows:

- a collection of objects to be handled uniformly but still have individual differences
- the behavior of an object to be easily changed by changing a strategy object
- an alternative implementation to be used with a minimum number of changes to the client code

Polymorphism plays a significant role in the implementation and execution of methods inherited from the `Object` class, including `toString`, `equals`, and `clone`. The strategy pattern is used extensively in laying out graphical user interfaces.



12.9 Problem Set

Written Exercises

12.1 The `move` method in the `LeftDancer` class (see Listing 12-1) contains the statement `super.move()` (lines 16, 18, and 20). What would happen if one of those statements were `this.move()`?

12.2 Polymorphism is like a ship's commanding officer yelling, "Battle stations!" Each member of the crew knows exactly what he should do in response to that order—and does it. The commander doesn't need to give each crew member individual instructions. Think of three more real-life analogies for polymorphism.

12.3 Write `comparator` classes that can be used to sort an array of:

a. `Robot` objects in ascending order by distance from the origin

b. `LineItem` objects in descending order by unit cost

c. `Person` objects (see Section 10.1) by role. Persons with the same role should be ordered by gender, while persons with the same role and gender should be ordered by decreasing age. (*Hint*: Use `compareTo` to compare enumerations.)

12.4 Draw a class diagram for the drawing program example in Section 12.1.3.

12.5 Study the documentation for the specified class. Draw a partial class diagram of it and its subclasses, showing the most important overridden methods and additional features of each subclass.

a. `becker.robots.Sim`

b. `java.text.Format`

c. `java.awt.Component` (This is the root of a huge hierarchy. Stop when your diagram includes about 10 classes, some of which are at least sub-subclasses of `Component`.)

d. `java.io.Reader` (Include a brief description of the functionality each subclass adds.)

12.6 Read the documentation for the `Box` class. What combination of classes does it replace? Describe what "struts" and "glue" are and how they might be used.

Programming Exercises

12.7 In the dancing robots example, it appears the fundamental difference between a `LeftDancer` and a `RightDancer` is not in how they move but in their favored direction to turn. Refactor the dancing robots example shown in Figure 12-3 so that `move` and `pirouette` are completely defined in the abstract class in terms of `turn` and `antiTurn`. These last two methods are abstract and must be overridden in both `LeftDancer` and `RightDancer`.

12.8 Investigate the documentation for `becker.robots.IPredicate`. For each of the following, write the predicate and a simple robot test program.

a. Write a predicate to identify a `Streetlight` that is on. Use it to turn off several streetlights.

b. The `City` class has a method named `setThingCountPredicate`. If `showThingCounts` is set to `true`, the number of things on each intersection that meet the predicate's criteria will be shown. The default counts the number of things that can be moved by a robot. Change it to show the total of all things, except robots.

c. Extend `Robot` to include a query, `northIsBlocked`, which returns `true` if the north exit to the intersection is blocked by a `Thing` such as a `Wall`. The query will use `isBesideThing` and a predicate that you write. The robot should not turn while executing this query.

d. Extend `Robot` to include a query, `dirIsBlocked(int dir)`. It is similar to `northIsBlocked` in part (c), but is not restricted to a single direction. The predicate will need an instance variable to remember the direction. The robot should not turn while executing this query.

12.9 Consider a family of robots that all have a `doMyThing` method. When a baby robot does its thing, it moves in a random direction with a random speed. Parent robots do their thing by moving and automatically picking up all the things found on their new intersection. Grandparent robots do their thing by moving at one-third the speed of a normal robot.

a. Implement the robot family by extending `RobotSE` three times. Write a `main` method containing an array of family members. Also scatter a number of `Thing` objects around. Make each robot do its thing 10 times. (*Hint*: You will need to introduce an abstract class.)

b. Implement the robot family by writing `FamilyMemberBot`. It extends `RobotSE` to use an instance of `IMoveStrategy`. Write the interface `IMoveStrategy` and three classes that implement it. Write a `main` method containing an array of `FamilyMemberBots`. Also scatter a number of `Thing` objects around. Make each robot do its thing five times. Change each robot to use a different move strategy, and then move each robot five more times. (*Hint*: The method in `IMoveStrategy` will take an instance of `FamilyMemberBot`, named `bot`, as a parameter and could contain method calls like `bot.move()`).

12.10 Some courses assign letter grades, whereas other courses assign a percentage between 0 and 100. Still others assign a pass/fail grade.

Write an interface named `Grade`. The `toPercent` method returns the grade as an integer percentage between 0 and 100 percent. The `toString` method prints the grade in its "native" format (a percentage, a letter grade, or either "Pass" or "Fail"). The `isPass` method returns `true` for a passing grade, `false` otherwise. The `includeInAverage` returns `true` for letter and numeric grades, but `false` for pass/fail grades.

Write three classes that implement `Grade`: `LetterGrade`, `PercentageGrade`, and `PassFailGrade`. Write a `main` method that fills an array with grades. For each grade, print on one line the native format, "Pass" or "Fail" (as appropriate), and the percentage (if it can be included in an average). After the list of grades, print the average grade as a percentage.

Use your school's mapping between letter grades and numeric grades, if it has one. Otherwise, make up something like `A+` is 95%, `A` is 90%, etc.

12.11 Write a `main` method that displays a `JPanel` inside a `JFrame` to arrange components as follows.

a. Use a `GridLayout` to arrange `JCheckBox` and `JSlider` components as shown in Figure 12-23a.

b. Use a combination of `BorderLayout`, `BoxLayout`, and `FlowLayout` to arrange `JRadioButton`, `JButton`, and `JTextArea` components as shown in Figure 12-23b. The text field will have no size unless you specify the rows and columns when it is created.

c. Approximate (b) as closely as you can using only `BoxLayout` and `FlowLayout`. You may find calling `setAlignmentY(0.0F)` on one of the panels useful.

d. Approximate (b) as closely as you can using only `GridBagLayout` and `FlowLayout`. You will need to read the `GridBagConstraints` class documentation carefully.

(figure 12-23)

Possible layouts

a) One layout

b) Another layout

12.12 Finish the program in Listing 12-22 so that it also displays Forfeit and New Game buttons, as shown in Figure 12-21. Include a `main` method that displays `HangmanView` in a `JFrame`. You won't be able to play a game with your program, but it should look good.

For an additional challenge, read about the `Box` class and figure out how to use it to replace a `JPanel` organized with a `BoxLayout` strategy.

12.13 In Section 10.1.5, we discussed various operations on those elements of an array that satisfy a specified property. For example, calculate the average age of everyone who is a "Little" or print all the people who are "Bigs."

Download the Big Brother/Big Sister example from Chapter 10 (`ch10/bbbs/`). Add an interface, `IInclude`, which has a single method, `boolean include(Person p)`. Add the following methods to `BigBroBigSis.java`:

a. `int countSatisfy(IInclude include)` counts those persons who satisfy `include`.

b. `double averageAge(IInclude include)` finds the average age of all those people who satisfy `include`.

c. `void list(IInclude include, PrintWriter out)` lists to the specified file all those people who satisfy `include`.

d. `Person[] subset(IInclude include)` returns a filled array of all those people who satisfy `include`.

Write a `main` method to test your methods using an instance of `IInclude` that specifies female "Bigs."

Programming Projects

12.14 Implement a simple bank application. The bank will have many accounts, each with an account number and a balance. A command interpreter will allow customers to enter one of the following commands:

- `d xxx yyy` (deposits the amount `xxx` to account number `yyy`).
- `w xxx yyy` (withdraws the amount `xxx` from account `yyy`).
- `t xxx yyy zzz` (withdraws the amount `xxx` from account `yyy` and deposits the same amount in account `zzz`).
- `b yyy` (displays the balance of account `yyy`).

The bank has two kinds of accounts. A `PerUseAccount` charges a set fee of $0.50 for each withdrawal. A `MinBalanceAccount` charges a fee of $1.00 for each withdrawal if the balance is less than $1,000. If the balance is $1,000 or more, no fee is charged.

a. Implement the banking system *without* using polymorphism or inheritance. Write a brief document outlining in point form what would have to be done to add a new kind of bank account to the system.

b. Implement the banking system using an inheritance hierarchy for the account classes. Take advantage of polymorphism but minimize the use of casting. Write a brief document outlining in point form what would have to be done to add a new kind of bank account to the system.

12.15 Implement a simple guessing game in which the user chooses a number that the program will try to guess. After each guess, the user will answer with either `H` (the guess was too high), `L` (the guess was too low), or `C` (the guess was correct). Allow the user to easily change the guessing strategy used by the program at the beginning of each game. Strategies should include at least two of the following:

a. Guess a random number.

b. Guess a number that is one larger than the previous guess. The first guess should be the smallest legal number for the game.

c. Guess the smallest legal number. As long as the user responds with `L`, guess a number that is 10 larger than the previous guess. When the user says it's too large, start guessing a number that is one less than the previous guess.

d. Based on the user's answers, keep track of the upper and lower limits on the number that could have been chosen by the user. Each guess should be the average of these two values. After each guess, update the upper and lower limits. (This brief description describes how most people search a physical phonebook: start in the middle and successively eliminate half of the remaining entries.)

12.16 Extend `JComponent` to paint shapes on itself. The shapes that it paints and their locations will depend on which shapes are added to a list the component maintains. Your shapes should form an inheritance hierarchy with `Shape` at the root. `Shape` should extend `Object`.

a. Demonstrate your program with a `main` method that adds a number of rectangles, circles, lines, and stars to your subclass of `JComponent`.

b. Implement two additional shapes of your choice beyond those required in (a).

c. Enhance your program so that it will paint the shapes it reads from a file.

12.17 Implement a simplified game of Monopoly that has an inheritance hierarchy of `BoardSquare` objects. Subclasses must include `Property`, `Railroad`, `Go`, and `IncomeTax`. You will also need a `Player` class and a command interpreter to play the game.

Think carefully about whether the `Player` object should react according to the kind of square it landed on or whether the `BoardSquare` objects should react to `Players` landing on or crossing them.

a. Implement the game with two instances of `Player` that always ask a user for what to do.

b. Implement the game to allow between two and six players, each of which uses a `move` strategy object to determine how it plays. Provide at least three different strategies: one that asks the user, another that always buys a property if it can, and a third that only buys a property if it has at least $500. Give the user a choice of strategies for each player when the game begins.

12.18 ACME Inc. has a standing order for 50 widgets each week from XYZ Inc. The agreement is that ACME sends the widgets each Friday and XYZ will send a check to pay for them that same day. If both live up to their agreement, they both profit. On the other hand, XYZ might send a fraudulent check, hoping to receive goods for free; or ACME might not send the goods, hoping to receive unearned payment.

We'll say either company "cooperates" if it abides by its side of the agreement. If it does not, we'll say the company "defects." The payoff can then be represented with Table 12-2.

ACME's Action	XYZ's Action	Value to ACME	Value to XYZ
Cooperate	Cooperate	3	3
Cooperate	Defect	-2	5
Defect	Cooperate	5	-2
Defect	Defect	0	0

(table 12-2)
Company actions

If both companies want to maximize their profit, what should their strategies be? Cooperate all of the time? Cooperate most of the time but defect occasionally? Cooperate as much as the other company cooperates?

First, develop three strategies that implement the following interface. They might be as simple as always cooperating, cooperating with the same probability that the other player has cooperated in the past, repeating the other player's last decision, or always defecting.

```
public interface ICommerceStrategy
{ public static final int DEFECT = 0;
  public static final int COOPERATE = 1;

  /** Decide whether to cooperate with the other player, given the other's history of
   *  cooperating with this player.
   *  @param other      The decisions made by the other player in previous turns. Each
   *                    element of the array is one of {DEFECT, COOPERATE}.
   *  @param numTurns   The number of turns made (other is partially filled).
   *  @return one of {DEFECT, COOPERATE} */
  public int getDecision(int[] other, int numTurns);
}
```

Second, develop a program that plays each strategy against all the other strategies, including a copy of itself. Print the cumulative score for each strategy to determine the best one. Assume that the players do not know how many turns there will be. (Does it change your strategy if you know this is your last turn?)