Chapter 2 Extending Classes with Services

Chapter Objectives

After studying this chapter, you should be able to:

- Extend an existing class with new commands
- Explain how a message sent to an object is resolved to a particular method
- Use inherited services in an extended class
- Override services in the superclass to provide different functionality
- Follow important stylistic conventions for Java programs
- Extend a graphical user interface component to draw a scene
- Add new kinds of `Things` and `Robots` to the robot world

Sometimes an existing class already does almost all of what is needed—but not quite all. For example, in the previous chapter, you may have found it unnatural to write `karel.turnLeft()` three times just to make a robot turn right. By extending a class we can add related services such as `turnRight`, allowing programmers to express themselves using language that better fits the problem.

Other approaches to providing new functionality include modifying the class itself to include the required functionality or writing a completely new class. These two approaches will be considered in later chapters.

2.1 Understanding Programs: An Experiment

Let's try an experiment. Find a watch or a clock that can measure time in seconds. Measure the number of seconds it takes you to understand the program shown in Listing 2-1. In particular, describe the path the robot takes, its final position, and its direction.

FIND THE CODE

ch02/experiment/

Listing 2-1: *An experiment in understanding a longer program*

```java
import becker.robots.*;

public class Longer
{
  public static void main(String[] args)
  { City austin = new City();
    Robot lisa = new Robot(austin, 3, 3, Direction.EAST);

    lisa.move();
    lisa.move();
    lisa.move();
    lisa.turnLeft();
    lisa.turnLeft();
    lisa.turnLeft();
    lisa.move();
    lisa.move();
    lisa.move();
    lisa.turnLeft();
    lisa.turnLeft();
    lisa.move();
    lisa.move();
    lisa.move();
    lisa.turnLeft();
    lisa.move();
    lisa.move();
    lisa.move();
    lisa.turnLeft();
    lisa.turnLeft();
  }
}
```

Now, imagine that we had a new kind of robot with commands to turn around, turn right, and move ahead three times. Time yourself again while you try to understand the program in Listing 2-2. The robot in this program does something different. How fast can you accurately figure out what?

FIND THE CODE

ch02/experiment/

Listing 2-2: *An experiment in understanding a shorter program*

```
import becker.robots.*;

public class Shorter
{
  public static void main(String[] args)
  { City austin = new City();
    ExperimentRobot lisa = new ExperimentRobot(
                                    austin, 3, 2, Direction.SOUTH);

    lisa.move3();
    lisa.turnRight();
    lisa.move3();
    lisa.turnAround();
    lisa.move3();
    lisa.turnLeft();
    lisa.move3();
    lisa.turnAround();
  }
}
```

You probably found the second program easier—and faster—to read and understand. The results shown in Table 2-1 are from a group of beginning Java programmers who performed the same experiment.

(table 2-1)

Results of an experiment in understanding programs

Program	Minimum Time (seconds)	Average Time (seconds)	Maximum Time (seconds)
Longer	12	87	360
Shorter	10	46	120

KEY IDEA

Adapt your language to express your ideas clearly and concisely

Why did we comprehend the second program more quickly? We raised the level of abstraction; the language we used (`turnAround`, `turnRight`, `move3`) matches our thinking more closely than the first program. Essentially, we created a more natural programming language for ourselves.

Raising the level of abstraction with language that matches our thinking has a number of benefits.

- Raising the level of abstraction makes it easier to write the program. It's easier for a programmer to think, "And then I want the robot to turn around" than to think, "The robot should turn around so I need to tell it to turn left and then turn left again." We can think of a task such as `turnAround`, deferring the definition of the task until later. Abstraction allows us to concentrate on the big picture instead of getting stuck on low-level details.
- Raising the level of abstraction allows us to understand programs better. An instruction such as `turnAround` allows the programmer to express her intent. Knowing the intent, we can better understand how this part of the program fits with the rest of the program. It's easier to be told that the programmer wants the robot to turn around than to infer it from two consecutive `turnLeft` commands.
- When we know the intent, it is easier to debug the program. Figuring out what went wrong when faced with a long sequence of service invocations is hard. When we know the intent, we can first ask if the programmer is intending to do the correct thing (**validation**), and then we can ask if the intent is implemented correctly (**verification**). This task is much easier than facing the entire program at once and trying to infer the intent from individual service invocations.

LOOKING AHEAD

Quality software is easier to understand, write, debug, reuse, and modify. We will explore this further in Chapter 11.

- We will find that extending the language makes it easier to modify the program. With the intent more clearly communicated, it is easier to find the places in the program that need modification.
- We can create commands that may be useful in other parts of the program, or even in other programs. By reusing old services and creating new services that will be easy to reuse in the future, we can save ourselves effort. We're working smarter rather than harder, as the saying goes.

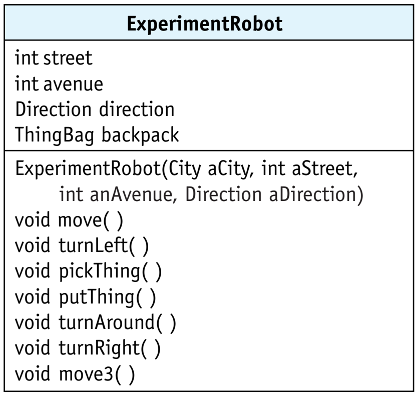
KEY IDEA

Work smarter by reusing code.

In the next section, we will learn how to extend an existing class such as `Robot` to add new services such as `turnAround`. We'll see that these ideas apply to all Java programs, not just those involving robots.

2.2 Extending the `Robot` Class

Let's see how the new kind of robot used in Listing 2-2 was created. What we want is a robot that can turn around, turn right, and move ahead three times—in addition to all the services provided by ordinary robots, such as turning left, picking things up, and putting them down. In terms of a class diagram, we want a robot class that corresponds to Figure 2-1.



```
ExperimentRobot
-----------------------------------------------
int street
int avenue
Direction direction
ThingBag backpack
-----------------------------------------------
ExperimentRobot(City aCity, int aStreet,
      int anAvenue, Direction aDirection)
void move( )
void turnLeft( )
void pickThing( )
void putThing( )
void turnAround( )
void turnRight( )
void move3( )
```

(figure 2-1)

Class diagram for the new robot class, `ExperimentRobot`

The class shown in Figure 2-1 is almost the same as `Robot`—but not quite. We would like a way to augment the existing functionality in `Robot` rather than implementing it again, similar to the camper shown in Figure 2-2.

(figure 2-2)

Van extended to be a camper

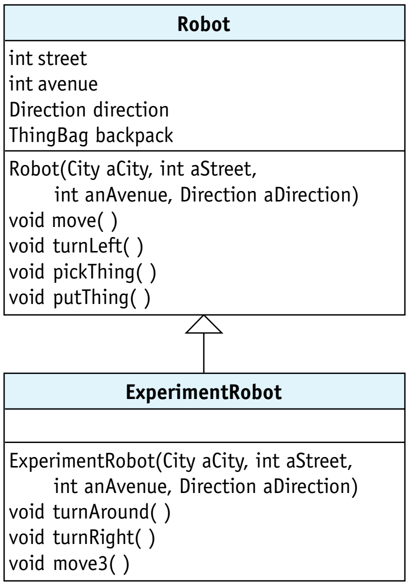
This vehicle has a number of features for people who enjoy camping: a bed in the pop-up top, a small sink and stove, a table, and so on. Did the camper's manufacturer design and build the entire vehicle just for the relatively few customers who want such a vehicle for camping? No. The manufacturer started with a simple cargo van and then added the special options for camping. The cargo van gave them the vehicle's basic frame, engine, transmission, driver's seat, instrument panel, and so on. Using all this infrastructure from an existing vehicle saved them a lot of work, so they could focus on the unique aspects required by a camper. The same cargo van, by the way, is also extended in different ways to carry more passengers.

Just as the camper manufacturer extended a van with additional features, we will extend `Robot` with additional services. Java actually uses the keyword `extends` for this purpose.

KEY IDEA

Start with something that does most of what you need. Then customize it for your particular use.

Figure 2-3 shows a class diagram in which `ExperimentRobot` extends `Robot`. Notice that the `Robot` class diagram is exactly the same as Figure 1-8. The `ExperimentRobot` class diagram shows only the attributes and services that are added to the `Robot` class. In the case of an `ExperimentRobot`, only services are added; no attributes. The hollow-tipped arrow between the two classes shows the relationship between them: `ExperimentRobot`, at the tail of the arrow, extends `Robot`, at the head of the arrow.



(figure 2-3)

Class diagram showing `ExperimentRobot` *extending* `Robot`

2.2.1 The Vocabulary of Extending Classes

When communicating about extending a class, we say that the `Robot` class is the **superclass** and the `ExperimentRobot` class is the **subclass**. Unless you're familiar with the language of mathematical sets or biology, this use of "sub" and "super" may seem backwards. In these settings, "super" means a more inclusive set or category. For example, an `ExperimentRobot` is a special kind of `Robot`. We will also define other special kinds of `Robots`. `Robot` is the more inclusive set, the superclass.

KEY IDEA

The class that is extended is called the "superclass." The new class is called the "subclass."

We might also say that `ExperimentRobot` **inherits** from `Robot` or that `ExperimentRobot` **extends** `Robot`.

If you think of a superclass as the parent of a class, that child class can have a grandparent and even a great-grandparent because the superclass may have its own superclass. It is, therefore, appropriate to talk about a class's superclasses (plural) even though it has only one direct superclass.

In a class diagram such as Figure 2-3, the superclass is generally shown above the subclass, and the arrow always points from the subclass to the superclass.

2.2.2 The Form of an Extended Class

LOOKING AHEAD

When a pattern is used, an icon appears in the margin, as shown here for the Extended Class pattern. The patterns for this chapter are revisited in Section 2.8.

PATTERN

Extended Class

The form of an extended class is as follows:

```
1  import «importedPackage»;
2
3  public class «className» extends «superClass»
4  {
5     «list of attributes used by this class»
6     «list of constructors for this class»
7     «list of services provided by this class»
8  }
```

The `import` statement is the same here as in the Java Program pattern. Line 3 establishes the relationship between this class and its superclass using the keyword `extends`. For an `ExperimentRobot`, for example, this line would read as follows:

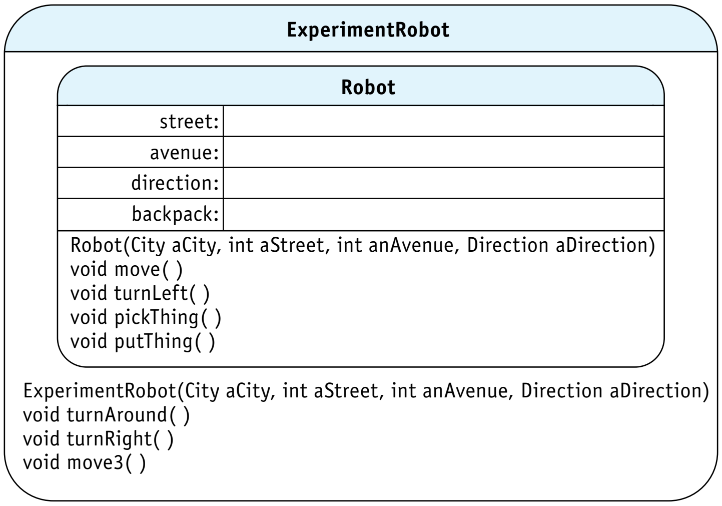
```
public class ExperimentRobot extends Robot
```

Lines 5, 6, and 7 of the code template are slots for attributes, constructors, and services. In the next section, we will implement a constructor. In the following sections, we will implement the services `turnAround`, `turnRight`, and `move3`. We will not be adding attributes to our classes until Chapter 6. Until then, we will use only the attributes inherited from the superclass.

2.2.3 Implementing a Constructor

The purpose of the constructor is to initialize each object that is constructed. That is, when the statement `Robot karel = new Robot(austin, 1, 1, Direction.SOUTH)` is executed, the constructor for the `Robot` class ensures that the attributes `street`, `avenue`, `direction`, and `backpack` are all given appropriate values.

We are not adding any attributes to the `ExperimentRobot`. So what is there to initialize? Is a constructor necessary? Yes. Because `ExperimentRobot` extends the `Robot` class, each `ExperimentRobot` object can be visualized as having a `Robot` object inside of it (see Figure 2-4). We need to ensure that the `Robot`-inside-the-`ExperimentRobot` object is correctly initialized with appropriate values for `street`, `avenue`, `direction`, and `backpack`.



(figure 2-4)

Visualizing an `ExperimentRobot` *as containing a* `Robot`

The constructor for `ExperimentRobot` is only four lines long—three if all the parameters would fit on the same line:

```
1  public ExperimentRobot(City aCity, int aStreet,
2                               int anAvenue, Direction aDirection)
3  { super(aCity, aStreet, anAvenue, aDirection);
4  }
```

Lines 1 and 2 declare the parameters required to initialize the `Robot`-inside-the-`ExperimentRobot`: a city, the initial street and avenue, and the initial direction. Each parameter is preceded by its type.

KEY IDEA

The constructor must ensure the superclass is properly initialized.

Line 3 passes on the information received from the parameters to the `Robot`-inside-the-`ExperimentRobot`. Object initialization is performed by a constructor, so you would think that line 3 would call the constructor of the superclass:

```
    Robot(aCity, aStreet, anAvenue, aDirection);        // doesn't work!
```

LOOKING AHEAD

Section 2.6 explains another use for the keyword `super`.

However, the designers of Java chose to use a keyword, `super`, instead of the name of the superclass. When `super` is used as shown in line 3, Java looks for a constructor in the superclass with parameters that match the provided arguments, and calls it. The effect is the same as you would expect from using `Robot`, as shown earlier.

LOOKING AHEAD

We will explore parameters further in Section 4.6 in the context of writing services with parameters.

When an `ExperimentRobot` is constructed with the following statement, the values passed as arguments (`austin`, `3`, `2`, and `Direction.SOUTH`) are copied into the parameters (`aCity`, `aStreet`, `anAvenue`, and `aDirection`) in the `ExperimentRobot` constructor.

```
ExperimentRobot lisa = new ExperimentRobot(austin,
                              3, 2, Direction.SOUTH);
```

Then, in line 3, those same values are passed as arguments to the parameters in the superclass's constructor.

Two other details about the constructor are that it must have the same name as the class and it does not have a return type—not even `void`. If a constructor has a name different from the class, the compiler considers it a service without a return type, and issues a compile-time error. If a constructor has a return type, the compiler considers it a service, and may not display an error until a client tries to use the constructor. Then the compiler will complain that it can't find it—because the constructor is being interpreted as a service.

Listing 2-3 contains the first steps in defining the `ExperimentRobot` class. It has a number of the template slots filled in, including imported classes, the class name, and the extended class's name. It also includes a constructor, but none of the new services. Just like the programs we wrote in Chapter 1, this class should be placed in its own file named `ExperimentRobot.java`.

FIND THE CODE

ch02/experiment/

Listing 2-3: *The* `ExperimentRobot` *class with a constructor but no services*

```
import becker.robots.*;

public class ExperimentRobot extends Robot
{
  public ExperimentRobot(City aCity, int aStreet,
                         int anAvenue, Direction aDirection)
  { super(aCity, aStreet, anAvenue, aDirection);
  }

  // The new services offered by an ExperimentRobot will be inserted here.
}
```

With this modest beginning, we can write a program that includes the following statement:

```
ExperimentRobot lisa = new ExperimentRobot(austin,
                              3, 2, Direction.SOUTH);
```

The robot `lisa` can do all things any normal robot can do. `lisa` can move, turn left, pick things up, and put them down again. An `ExperimentRobot` is a kind of `Robot` object and has inherited all those services from the `Robot` class. In fact, the `Robot` in line 7 of Listing 2-1 could be replaced with an `ExperimentRobot`. Even with no other changes, the program would execute as it does with a `Robot`. However, an `ExperimentRobot` cannot yet respond to the messages `turnAround`, `turnRight`, or `move3`.

KEY IDEA

A class with only a constructor, like Listing 2-3, is complete and can be used in a program. It just doesn't add any new services.

2.2.4 Adding a Service

A service is an idea such as "turn around." To actually implement this idea, we add a **method** to the class, which contains code to carry out the idea. When we want a robot to turn around, we send a message to the robot naming the `turnAround` service. This message causes the code in the corresponding method to be executed.

KEY IDEA

Services are ideas. Methods contain code to implement the idea. Services are invoked with a message. The object responds by executing a method.

An analogy may help distinguish services, messages, and methods. Every child can eat. This is a service provided by the child. It is something the child can do. A message from a parent, "Come and eat your supper" causes the child to perform the service of eating. The particular method the child uses to eat, however, depends on the instructions he or she has received while growing up. The child may use chopsticks, a fork, or a fork and a knife. The idea (eating) is the service. The message ("eat your supper") causes the service to be performed. How the service is performed (chopsticks, fork, and so on) is determined by the instructions in the method.

The service `turnAround` may be added to the `ExperimentRobot` class by inserting the following method between lines 10 and 11 in Listing 2-3:

```
public void turnAround()
{ this.turnLeft();
  this.turnLeft();
}
```

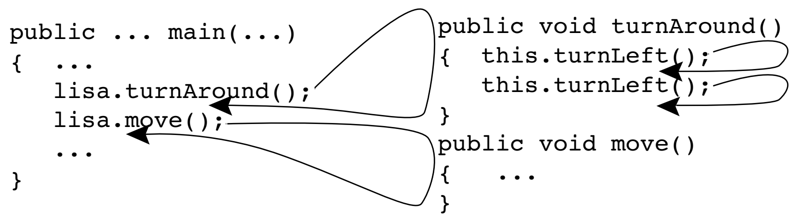
Now the robot `lisa` can respond to the `turnAround` message. When a client says `lisa.turnAround()`, the robot knows that `turnAround` is defined as turning left twice, once for each `turnLeft` command in the body of `turnAround`.

Flow of Control

Recall the Sequential Execution pattern from Chapter 1. It says that each statement is executed, one after another. Each statement finishes before the next one in the sequence begins. When a program uses `lisa.turnAround()` we break out of the Sequential Execution pattern. The **flow of control**, or the sequence in which statements are executed, does not simply go to the next statement (yet). First it goes to the statements contained in `turnAround`, as illustrated in Figure 2-5.

(figure 2.5)

Flow of control when executing methods



```
public ... main(...)
{  ...
   lisa.turnAround();
   lisa.move();
   ...
}
```

```
public void turnAround()
{  this.turnLeft();
   this.turnLeft();
}
public void move()
{   ...
}
```

When main sends the message `lisa.turnAround()`, Java finds the definition of `turnAround` and executes each of the statements it contains. It then returns to the statement following `lisa.turnAround()`. This is an example of a much more general pattern that occurs each time a message is sent to an object:

KEY IDEA

Calling a method temporarily interrupts the Sequential Execution pattern.

- The method implementing the service named in the message is found.
- The statements contained in the method are executed. Unless told otherwise, the statements are executed sequentially according to the Sequential Execution pattern.
- Flow of control returns to the statement following the statement that sent the message.

Look again at Figure 2-5. This same pattern is followed when `lisa` is sent the `move` message, although we don't know what the statements in the `move` method are, so they are not shown in the figure. Similarly, when `turnAround` is executed, each `turnLeft` message it sends follows the same pattern: Java finds the method implementing `turnLeft`, executes the statements it contains, and then it returns, ready to execute the next statement in `turnAround`.

When we are considering only `main`, the Sequential Execution pattern still holds. When we look only at `turnAround`, the Sequential Execution pattern holds there, too. But when we look at a method together with the methods it invokes, we see that the Sequential Execution pattern does *not* hold. The flow of control jumps from one place in the program to another—but always in an orderly and predictable manner.

The Implicit Parameter: this

In previous discussions, we have said that parameters provide information necessary for a method or constructor to do its job. Because `turnAround` has no parameters, we might conclude that it doesn't need any information to do its job. That conclusion is not correct.

One vital piece of information `turnAround` needs is which robot it should turn around. When a client says `lisa.turnAround()`, the method must turn `lisa` around, and if a client says `karel.turnAround()`, the method must turn `karel` around. Clearly the method must know which object it is to act upon.

This piece of information is needed so often and is so vital that the designers of Java made accessing it extremely easy for programmers. Whenever a method is invoked with the pattern *«objectReference».«methodName»(…)*, *«objectReference»* becomes an **implicit parameter** to *«methodName»*. The implicit parameter is the object receiving the message. In the case of `lisa.turnAround()`, the implicit parameter is `lisa`, and for `karel.turnAround()`, the implicit parameter is `karel`.

The implicit parameter is accessed within a method with the keyword `this`. The statement `this.turnLeft()` means that the same robot that called `turnAround` will be instructed to turn left. If the client said `lisa.turnAround()`, then `lisa` will be the implicit parameter and `this.turnLeft()` will instruct `lisa` to turn left.

Sometimes when a person learns a new activity with many steps they will mutter instructions to themselves: "First, *I* turn left. Then *I* turn left again." Executing a method definition is like that, except that "I" is replaced by "this robot." You can think of the `ExperimentRobot` as muttering instructions to itself: "First, *this* robot turns left. Then *this* robot turns left again."

`public` and `void` Keywords

The two remaining keywords in the method definition are `public` and `void`. The keyword `public` says that this method is available for any client to use. In Section 3.6, we will learn about situations for which we might want to prevent some clients from using certain methods. In those situations, we will use a different keyword.

The keyword `void` distinguishes a command from a query. Its presence tells us that `turnAround` does not return any information to the client.

2.2.5 Implementing `move3`

Implementing the `move3` method is similar to `turnAround`, except that we want the robot to move forward three times. The complete method follows. Like `turnAround`, it is placed inside the class, but outside of any constructor or method.

```
public void move3()
{ this.move();
  this.move();
  this.move();
}
```

As with `turnAround`, we want the same robot that is executing `move3` to do the moving. We therefore use the keyword `this` to specify which object receives the `move` messages.

LOOKING AHEAD

Eventually we will learn how to write a method with a parameter so we can say `lisa.move(50)`*—or any other distance.*

2.2.6 Implementing `turnRight`

To tell a robot to turn right, we could say "turn left, turn left, turn left." We could also say "turn around, then turn left." Both work. The first approach results in the following method:

PATTERN

Parameterless Command

```
public void turnRight()
{ this.turnLeft();
  this.turnLeft();
  this.turnLeft();
}
```

KEY IDEA

An object can send messages to itself, invoking its own methods.

The second approach is more interesting, resulting in this method:

```
public void turnRight()
{ this.turnAround();
  this.turnLeft();
}
```

LOOKING AHEAD

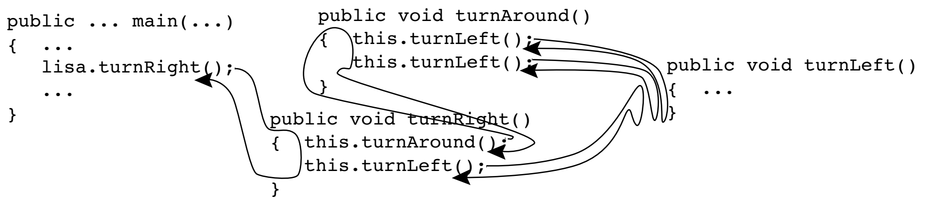
Methods calling other methods is a core idea of Stepwise Refinement, the topic of Chapter 3.

The second version works by asking the `ExperimentRobot` object to execute one of its own methods, `turnAround`. The robot finds the definition of `turnAround` and executes it (that is, it turns left twice as the definition of `turnAround` says it should). When it has finished executing `turnAround`, it is told to `turnLeft` one more time. The robot has then turned left three times, as desired.

This flow of control is illustrated in Figure 2-6. Execution begins with `lisa.turnRight()` in the `main` method. It proceeds as shown by the arrows. Each method executes the methods it contains and then returns to its client, continuing with the statement after the method call. Before a method is finished, each of the methods it calls must also be finished.

(figure 2-6)

Flow of control when one method calls another


```
public ... main(...)
{   ...
    lisa.turnRight();
    ...
}

public void turnAround()
{   this.turnLeft();
    this.turnLeft();
}

public void turnRight()
{   this.turnAround();
    this.turnLeft();
}

public void turnLeft()
{   ...
}
```

This discussion completes the `ExperimentRobot` class. The entire program is shown in Listing 2-4.

Listing 2-4: *The complete listing for* `ExperimentRobot`

```
import becker.robots.*;

// A new kind of robot that can turn around, turn right, and move forward
// three intersections at a time.
// author: Byron Weber Becker
public class ExperimentRobot extends Robot
{
  // Construct a new ExperimentRobot.
  public ExperimentRobot(City aCity, int aStreet,
                          int anAvenue, Direction aDirection)
  { super(aCity, aStreet, anAvenue, aDirection);
  }

  // Turn this robot around so it faces the opposite direction.
  public void turnAround()
  { this.turnLeft();
    this.turnLeft();
  }

  // Move this robot forward three times.
  public void move3()
  { this.move();
    this.move();
    this.move();
  }

  // Turn this robot 90 degrees to the right by turning around and then left by 90 degrees.
  public void turnRight()
  { this.turnAround();
    this.turnLeft();
  }
}
```

FIND THE CODE

ch02/experiment/

2.2.7 RobotSE

You can probably imagine other programs requiring robots that can turn around and turn right. `DeliverParcel` (Listing 1-1) could have used `turnRight` in one place, while `GoAroundRoadBlock` (Listing 1-2) could have used it twice. Several of the programming projects at the end of Chapter 1 could have used either `turnAround` or `turnRight` or both.

LOOKING AHEAD

You learn how to build your own package of useful classes in Chapter 9.

When we write methods that are applicable to more than one problem, it is a good idea to add that method to a class where it can be easily reused. The `becker` library has a class containing commonly used extensions to `Robot`, including `turnRight` and `turnAround`. It's called `RobotSE`, short for "Robot Special Edition." In the future, you may want to extend `RobotSE` instead of `Robot` so that you can easily use these additional methods.

2.2.8 Extension vs. Modification

LOOKING AHEAD

The `turnRight` method in `RobotSE` actually turns right instead of turning left three times. Section 3.6 explains how.

Another approach to making a robot that can turn around and turn right would be to modify the existing class, `Robot`. Modifying an existing class is not always possible, and this is one of those times. The `Robot` class is provided in a library, without source code. Without the source code, we have nothing to modify. We say that the `Robot` class is **closed for modification**.

There are other reasons to consider a class closed for modification, even when the source code is available. In a complex class, a company may not want to risk introducing errors through modification. Or the class may be used in many different programs, with only a few benefiting from the proposed modifications.

As we've seen, however, the `Robot` class is **open for extension**. It is programmed in such a way that those who want to modify its operation can do so via Java's extension mechanism. When a class is open for extension it can be modified via subclasses without fear of introducing bugs into the original class or introducing features that aren't generally needed.

2.3 Extending the `Thing` Class

KEY IDEA

Classes other than `Robot` can also be extended.

`Robot` is not the only class that can be extended. For example, `City` is extended by `MazeCity`. Instances of `MazeCity` contain a maze for the robots to navigate. At the end of this chapter you will find that graphical user interface components can be extended to do new things as well. In fact, every class can be extended unless its programmer has taken specific steps to prevent extension.

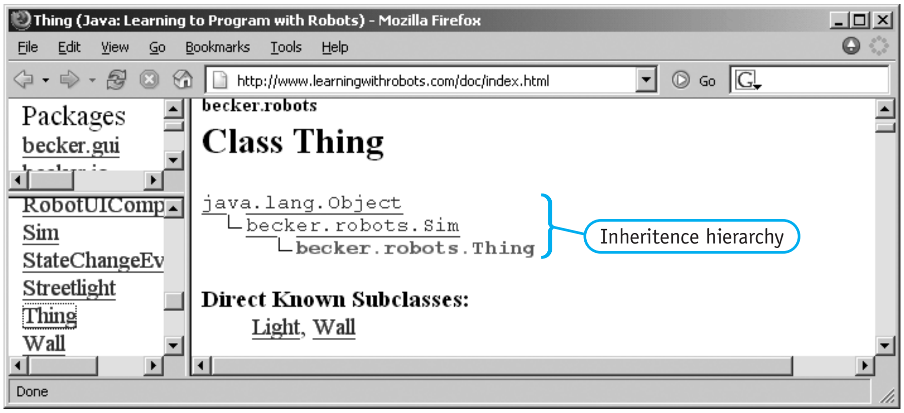
To demonstrate extending a class other than `Robot`, let's extend `Thing` to create a `Lamp` class. Each `Lamp` object will have two services, one to turn it "on" and another to turn it "off." When a lamp is "on" it displays itself with a soft yellow circle and when it is "off" it displays itself with a black circle. Because `Lamp` extends `Thing`, lamps behave like things—robots can pick them up, move them, and put them down again.

2.3.1 Exploring the `Thing` Class

Before attempting to extend the `Thing` class we should become more familiar with it. The beginning of its online documentation is shown in Figure 2-7. Part of the information it provides is the classes `Thing` extends, also called the **inheritance hierarchy**. In this case, `Thing` extends a class named `Sim`, and `Sim` extends `Object`. The `Sim` class defines core methods inherited by everything displayed in a city, including intersections, robots, and things. Below the inheritance hierarchy we are told that `Thing` is extended by at least two classes: `Light` and `Wall`.

LOOKING AHEAD

The `Light` subclass of `Thing` may be useful for defining lamps. We'll explore this idea later, in Section 2.3.6.


Thing (Java: Learning to Program with Robots) - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.learningwithrobots.com/doc/index.html
Go
Packages
becker.gui
RobotUIComp
Sim
StateChangeEv
Streetlight
Thing
Wall
Done
Inheritence hierarchy

becker.robots

Class Thing

```
java.lang.Object
  └becker.robots.Sim
      └becker.robots.Thing
```

Direct Known Subclasses:

Light, Wall

(figure 2-7)

Part of the documentation for the `Thing` class

The summaries for `Thing`'s four constructors are shown in Figure 2-8. The constructor we have used so far provides a "default appearance," which we know from experience is a yellow circle. The second and third constructors tell us that `Thing` objects have other properties such as whether they can be moved (presumably by a robot) and an orientation. The class overview (not shown in Figure 2-8) refers to these properties as well.

The methods listed in the documentation are mostly queries and don't seem helpful for implementing a `lamp` object. They are listed online, of course, and also in Appendix E. However, the `Thing` documentation also includes a section titled "Methods Inherited from Class `becker.robots.Sim`." This section lists a `setIcon` method. Its description says "Set the icon used to display this Sim."

Icons determine how each robot, intersection, or thing object appears within the city. By changing the icon, we can change how something looks. For example, the following few lines of code replace `deceptiveThing`'s icon to make the `Thing` look like a `Wall`:

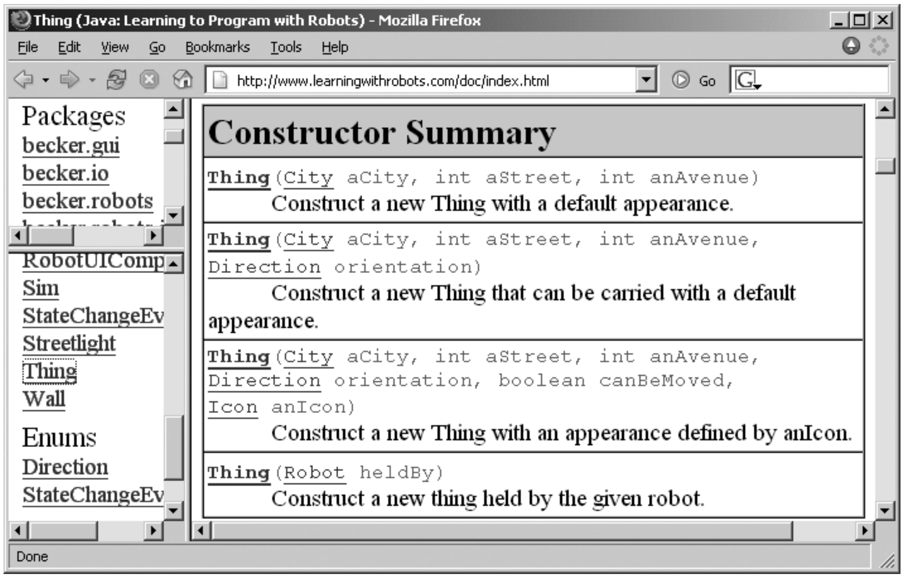
```
Thing deceptiveThing = new Thing(aCityObject, 3, 4);
WallIcon anIcon = new WallIcon();
deceptiveThing.setIcon(anIcon);
```

LOOKING BACK

The first two lines of code use the Object Instantiation pattern discussed in Section 1.7.2.

(figure 2-8)

Additional documentation for the `Thing` class


Thing (Java: Learning to Program with Robots) - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.learningwithrobots.com/doc/index.html
Go
Packages
becker.gui
becker.io
becker.robots
RobotUIComp
Sim
StateChangeEv
Streetlight
Thing
Wall
Enums
Direction
StateChangeEv
Done

Constructor Summary

`Thing(City aCity, int aStreet, int anAvenue)`
Construct a new Thing with a default appearance.

`Thing(City aCity, int aStreet, int anAvenue, Direction orientation)`
Construct a new Thing that can be carried with a default appearance.

`Thing(City aCity, int aStreet, int anAvenue, Direction orientation, boolean canBeMoved, Icon anIcon)`
Construct a new Thing with an appearance defined by anIcon.

`Thing(Robot heldBy)`
Construct a new thing held by the given robot.

We will do something similar for our `Lamp` class. When the lamp is turned on we will give it a new appearance using `setIcon`, passing it an icon that shows a soft yellow circle. When the lamp is turned off we will pass `setIcon` an icon showing a black circle.

2.3.2 Implementing a Simple `Lamp` Object

We will implement our `Lamp` class by extending `Thing`. Our experience with extending `Robot` tells us that to extend a class we must complete the following tasks:

- Use the `class` and `extends` keywords to specify the class's name and the name of the class it extends. (See Section 2.2.2.)
- Write a constructor that will initialize the superclass appropriately. (See Section 2.2.3.)
- Write methods implementing each of the services offered by the class. (See Section 2.2.4.)

Knowing these three things, we can write the beginnings of our `Lamp` class as shown in Listing 2-5. We still need to replace each ellipsis (…) with additional Java code.

FIND THE CODE

ch02/extendThing/

Listing 2-5: *The beginnings of a `Lamp` class*

```
import becker.robots.*;

```

Listing 2-5: *The beginnings of a* `Lamp` *class* (continued)

```
 3 public class Lamp extends Thing
 4 {
 5    // Construct a new lamp object.
 6    public Lamp(...)
 7    { super(...);
 8    }
 9
10    // Turn the lamp on.
11    public void turnOn()
12    { ...
13    }
14
15    // Turn the lamp off.
16    public void turnOff()
17    { ...
18    }
19 }
```

PATTERN

Extended Class

Implementing the Constructor

The `Lamp` constructor must ensure that the attributes in its superclass, `Thing`, are completely initialized when a lamp is created. Recall that attributes are initialized by calling one of the constructors in the `Thing` class using the keyword `super`. The arguments passed to `super` must match the parameters of a constructor in the superclass. The documentation in Figure 2-8 confirms that one of the constructors requires a city, initial street, and initial avenue as parameters. As with the `ExperimentRobot` class, this information will come via the constructor's parameters. Putting all this together, lines 6, 7, and 8 in Listing 2-5 should be replaced with the following code:

```
public Lamp(City aCity, int aStreet, int anAvenue)
{ super(aCity, aStreet, anAvenue);
}
```

PATTERN

Constructor

Implementing `turnOn` and `turnOff`

When the `turnOn` service is invoked, the lamp should display itself as a soft yellow circle. As we discovered earlier, the appearance is changed by changing the lamp's icon using the `setIcon` method inherited from a superclass.

LOOKING BACK

Finding robot documentation was discussed in Section 1.4.8.

The robot documentation (in the `becker.robots.icons` package) describes a number of classes that include the word "Icon" such as `RobotIcon`, `ShapeIcon`, `WallIcon`, `FlasherIcon`, and `CircleIcon`. The last one may be able to help display a yellow circle. According to the documentation, constructing a `CircleIcon` requires a `Color` object to pass as an argument. We'll need a `Color` object *before* we construct the `CircleIcon` object.

In summary, to change the appearance of our `Lamp` we must complete the following steps:

```
create a Color object named "onColor"
create a CircleIcon named "onIcon" using "onColor"
call setIcon to replace this lamp's current icon with "onIcon"
```

KEY IDEA

Documentation is useful. Bookmark it in your browser to make it easy to access.

We can learn how to construct a `Color` object by consulting the online documentation at *http://java.sun.com/j2se/1.5.0/docs/api/* or, if you have already found the documentation for `CircleIcon`, click on the link to `Color` found in the constructor's parameter list. The documentation describes seven `Color` constructors. The simplest one takes three numbers, each between 0 and 255, that specify the red, green, and blue components of the color. Using 255, 255, and 200 produces a soft yellow color appropriate for a lamp's light. A **color chooser** is a dialog that displays many colors and can help choose these three values. Most drawing programs have a color chooser and Problem 1.18 provides guidance for writing your own color chooser.

We can now convert the preceding steps to Java. Inserting them in the `turnOn` method results in the following five lines of code:

PATTERN

Parameterless Command

```
public void turnOn()
{  Color onColor = new Color(255, 255, 200);
   CircleIcon onIcon = new CircleIcon(onColor);
   this.setIcon(onIcon);
}
```

The `turnOff` method is identical except that `onColor` and `onIcon` should be appropriately named `offColor` and `offIcon`, and `offColor` should be constructed with `new Color(0, 0, 0)`.

Completing the Class and `Main` Method

We have now completed the `Lamp` class. Listing 2-6 shows it in its entirety. Notice that it includes two new import statements in lines 2 and 3. The first one gives easier[1] access to `CircleIcon`; it's in the `becker.robots.icons` package. The second one gives easier access to `Color`.

[1] It is possible to access these classes without the `import` statement. Every time the class is used, include the package name. For example, `java.awt.Color onColor = new java.awt.Color(255, 255, 200);`

Listing 2-6: *The* `Lamp` *class*

```
import becker.robots.*;
import becker.robots.icons.*;       // CircleIcon
import java.awt.*;                  // Color

public class Lamp extends Thing
{
  // Construct a new lamp object.
  public Lamp(City aCity, int aStreet, int anAvenue)
  { super(aCity, aStreet, anAvenue);
  }

  // Turn the lamp on.
  public void turnOn()
  { Color onColor = new Color(255, 255, 200);
    CircleIcon onIcon = new CircleIcon(onColor);
    this.setIcon(onIcon);
  }

  // Turn the lamp off.
  public void turnOff()
  { Color offColor = new Color(0, 0, 0);
    CircleIcon offIcon = new CircleIcon(offColor);
    this.setIcon(offIcon);
  }
}
```

FIND THE CODE

ch02/extendThing/

PATTERN

Extended Class

PATTERN

Parameterless Command

This example illustrates that many classes can be extended, not just the `Robot` class. To extend a class, we perform the following tasks:

- Create a new class that includes the following line:

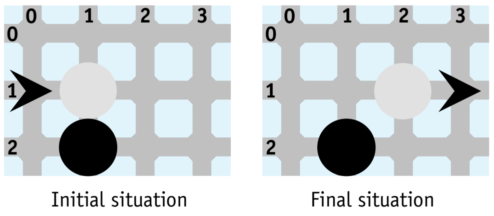
  `public class «className» extends «superClass»`

  where *«superClass»* names the class you want to extend. In this example the superclass was `Thing`; in the first example the superclass was `Robot`.
- Create a constructor for the new class that has the same name as the class. Make sure it calls `super` with parameters appropriate for one of the constructors in the superclass.
- Add a method for each of the services the new class should offer.

A short test program that uses the `Lamp` class is shown in Listing 2-7. It instantiates two `Lamp` objects, turning one on and turning the other off. A robot then picks one up and moves it to a new location. The left side of Figure 2-9 shows the initial situation after lines 7–14 have been executed. The right side of Figure 2-9 shows the result after lines 17–21 have been executed to move the lit lamp to another intersection. The actual running program is more colorful than what is shown in Figure 2-9.

(figure 2-9)

Program with two `Lamp` *objects, one "on" at (1, 1) and one "off" at (2, 1) in the initial situation*



FIND THE CODE

ch02/extendThing/

Listing 2-7: *A* `main` *method for a program that uses the* `Lamp` *class*

```
 1 import becker.robots.*;
 2
 3 public class Main
 4 {
 5   public static void main(String[] args)
 6   { // Construct the initial situation.
 7     City paris = new City();
 8     Lamp lamp1 = new Lamp(paris, 1, 1);
 9     Lamp lamp2 = new Lamp(paris, 2, 1);
10     Robot lampMover = new Robot(paris, 1, 0, Direction.EAST);
11
12     // Turn one lamp on and the other off.
13     lamp1.turnOn();
14     lamp2.turnOff();
15
16     // Use the robot to move one of the lamps.
17     lampMover.move();
18     lampMover.pickThing();
19     lampMover.move();
20     lampMover.putThing();
21     lampMover.move();
22   }
23 }
```

2.3.3 Completely Initializing Lamps

In Listing 2-7, the `main` method instantiates two `Lamp` objects in lines 8 and 9, and then explicitly turns one on and one off in lines 13 and 14. Suppose that lines 13 and 14 were omitted, so that the lamps were turned neither on nor off explicitly. How would they appear? Unfortunately, they would appear just like any other `Thing`—as a medium-sized, bright yellow circle. It seems wrong that a `Lamp` object should appear to be a `Thing` just because the client forgot to explicitly turn it on or off.

The problem is that the `Lamp` constructor did not completely initialize the object. A complete initialization for a lamp not only calls `super` to initialize the superclass, it also sets the icon so the lamp appears to be either on or off.

A constructor can execute statements other than the call to `super`. It could, for example, call `setIcon`. But to follow the Service Invocation pattern of *`«objectReference».«serviceName»(…)`*, we need to have an object. The object we want is the object the constructor is now creating.

The methods we have written often use the implicit parameter, `this`, to refer to the object executing the method. The implicit parameter, `this`, is also available within the constructor. It refers to the object being constructed. We can write `this.setIcon(…);` within the constructor. Think of `this` as referring to *this* object, the one being constructed.

LOOKING BACK

The implicit parameter was discussed in more depth in Section 2.2.4.

The new version of the constructor is then as follows. It initializes the superclass with the call to `super`, and then finishes the initialization by replacing the icon with a new one.

```
1 public Lamp(City aCity, int aStreet, int anAvenue)
2 { super(aCity, aStreet, anAvenue);
3   Color offColor = new Color(0, 0, 0);
4   CircleIcon offIcon = new CircleIcon(offColor);
5   this.setIcon(offIcon);
6 }
```

Calling `super` must be the first statement in the constructor. It ensures that the `Thing`-inside-the-`Lamp` is appropriately initialized so it can handle the call to `this.setIcon`.

KEY IDEA

Initializing the superclass is the first thing to do inside a constructor.

You may recognize lines 3–5 as being identical to the body of `turnOff`. Do we really need to type in the code twice, and then fix it twice if we discover a bug or want to change how a `Lamp` looks when it is off? No. Recall that an object can call its own methods. We saw this concept when `ExperimentRobot` called `turnAround` from the `turnRight` method. Similarly, the constructor can call `turnOff` directly, a much better solution.

```
public Lamp(City aCity, int aStreet, int anAvenue)
{ super(aCity, aStreet, anAvenue);
  this.turnOff();
}
```

PATTERN

Constructor

2.3.4 Fine-Tuning the `Lamp` Class (optional)

The `Lamp` class can be fine-tuned in several ways to be more visually pleasing.

Changing the Size of an Icon

A turned-off `Lamp` appears as large as the yellow circle of light cast by a lamp that is on, which is not realistic. It's also unrealistic to represent a lamp with an icon that is as large as an intersection.

To solve this problem, we need a way to make a smaller icon. The documentation for `CircleIcon` includes a method, `setSize`, for this purpose. Its parameter is a number between 0.0 and 1.0. A value of 1.0 makes it full size, 0.001 makes it extremely small, and 0.5 makes it half size. The size must be larger than 0.0.

With this knowledge, let's change `turnOff` to make a smaller icon:

PATTERN

Parameterless Command

```
public void turnOff()
{ Color offColor = new Color(0, 0, 0);
  CircleIcon offIcon = new CircleIcon(offColor);
  offIcon.setSize(0.25);
  this.setIcon(offIcon);
}
```

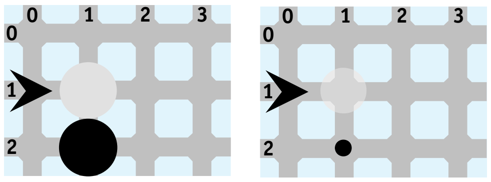
The only change in this code, compared to Listing 2-6, is to add the extra method call.

Transparency

To make the lamp more realistic, the light from a lamp should be semi transparent to let the intersection show through. With the previous change to `turnOff` and a small change to `turnOn`, the initial situation will look like Figure 2-11 instead of Figure 2-10. Unfortunately, the difference is not as striking in print as it is on-screen in full color.

To obtain a transparent color, we can again use a service `CircleIcon` inherits from its superclass. `setTransparency` takes a number between 0.0 and 1.0 where a value of 0.0 is completely opaque and 1.0 is completely transparent. For the lamp icon a value of about 0.5 works well. The new version of `turnOn` follows:

```
public void turnOn()
{ Color onColor = new Color(255, 255, 200);
  CircleIcon onIcon = new CircleIcon(onColor);
  onIcon.setSize(0.75);
  onIcon.setTransparency(0.5);
  this.setIcon(onIcon);
}
```



(figure 2-10)

Original initial situation (left)

(figure 2-11)

New initial situation (right)

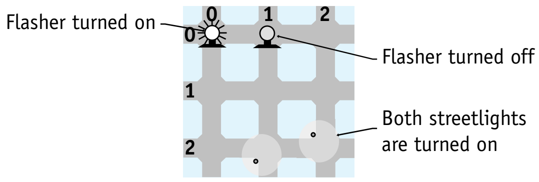
2.3.5 Other Subclasses of `Thing`

The `becker.robots` package includes several subclasses of `Thing` that have already been defined. They are similar to `Lamp`, except that someone else did the programming, and they have been put into the `becker.robots` package. Interested students may enjoy using them to give additional variety to their programs.

KEY IDEA

The `Thing` class can be extended in many ways.

One subclass of `Thing` is called `Flasher`. It represents the flashing lights used by road maintenance crews to mark hazards. A `Flasher` is like a `Lamp` except that when it is turned on, it flashes. An example is shown in Figure 2-12. The flasher at the origin is turned on. The flasher beside it is turned off.



(figure 2-12)

The appearance of `Flasher`s and `Streetlight`s

Another provided subclass of `Thing` is `Streetlight`. Two instances are shown at (2, 1) and (2, 2). Like walls, streetlights can occupy different positions on an intersection. These streetlights occupy the southwest and northwest corners of their respective intersections. They were created with the following code:

```
Streetlight sLight1 =
      new Streetlight(prague, 2, 1, Direction.SOUTHWEST);
Streetlight sLight2 =
      new Streetlight(prague, 2, 2, Direction.NORTHWEST);
```

Another similarity to walls is that streetlights cannot be picked up and carried by robots. One of the constructors to `Thing` includes a parameter that controls whether robots can pick the thing up; the `Streetlight` constructor makes use of that feature.

The streetlights shown in Figure 2-12 are turned on. Streetlights that are turned off show only the pole in the corner of the intersection. An intersection may have more than one streetlight.

2.3.6 Fun with Lights (optional)

We have already seen how lamps can be turned off and turned on in the `main` method (see Listing 2-7). The same can be done with `Flashers` and `Streetlights`. It is also possible for robots to turn lights on or off, provided the robot and the light are on the same intersection. This requires programming concepts that won't be fully explained until Section 12.1.5, but using them follows a simple pattern and offers more possibilities for creatively using robots.

The key is the `Light` class that we noticed in the documentation shown in Figure 2-7. It has two methods named `turnOn` and `turnOff` and is the superclass for `Flasher` and `Streetlight`. If we extend `Light` when we write the `Lamp` class, all three subclasses are, in some sense, lights that can be turned on and off.

LOOKING AHEAD

This is an example of polymorphism. Learn more in Chapter 12.

In a subclass of `Robot` we can use an inherited method, `examineLights`. It examines the robot's current intersection for a `Light` object. If it finds one, it makes it accessible to the robot. If a light does not exist on the intersection, an error will be printed and the program will stop. Listing 2-8 shows most of a subclass of `Robot` that makes use of `examineLights`. A `SwitchBot`'s `switchLights` method may be used where there are four intersections in a row, each one with a light on it.

FIND THE CODE

ch02/switchLights/

Listing 2-8: *A subclass of* `Robot` *that manipulates lights*

```java
import becker.robots.*;

// A robot that switches lights on and off.
public class SwitchBot extends Robot
{
   // Insert a constructor here.

   // Switch every other light off and the remaining lights on.
   public void switchLights()
   { this.move();
     this.examineLights().next().turnOff();
     this.move();
     this.examineLights().next().turnOn();
     this.move();
     this.examineLights().next().turnOff();
     this.move();
     this.examineLights().next().turnOn();
     this.move();
   }
}
```

Notice that lines 11, 13, 15, and 17 have three method calls. This is permitted when a method returns an object. That method call may then be followed with a call to another method. The second method call must be appropriate for the object returned by the first method.

The `Robot` class also has methods named `examineThings` and `examineRobots` that can be used similarly. The `Intersection` and `City` classes have similar methods available.

2.4 Style

As programs become more complex, presenting them clearly to anyone who reads them (including ourselves) becomes vitally important. Attention to presentation, choosing names wisely, indenting, and commenting code all contribute to a program's clarity. In the course of writing and debugging your programs, you will be studying them more thoroughly than anyone else. It's to *your* advantage to make your programs as understandable as possible.

KEY IDEA

Everyone, especially you, benefits from good programming style.

2.4.1 White Space and Indentation

White space is the empty space between the symbols in a program. The Java compiler ignores white space, but its presence (or absence) is important to people reading the program. Consider the program in Listing 2-9. It is identical to the `ExperimentRobot` class in Listing 2-4, except that the white space and comments have been removed. Both classes execute in exactly the same way, but one is considerably easier to read and understand than the other. In particular, Listing 2-9 makes it difficult to see the structure of the class: that there is one constructor and three methods, where each method begins, what the method names are, and so on.

KEY IDEA

Use white space to highlight the logical structure of your program.

Listing 2-9: *A class without white space*

```
import becker.robots.*; public class ExperimentRobot extends
Robot { public ExperimentRobot(City aCity, int aStreet, int
anAvenue, Direction aDirection) { super(aCity, aStreet,
anAvenue, aDirection);} public void turnAround() {
this.turnLeft(); this.turnLeft(); } public void move3(){
this.move(); this.move(); this.move(); } public void
turnRight() { this.turnAround(); this.turnLeft(); }}
```

The following recommendations concerning white space are considered good programming practice:

- Begin each statement on a new line.
- Include at least one blank line between blocks of code with different purposes. For instance, Listing 1-2 includes blank lines between the statements that construct the required objects and the statements that direct the robot `mark` around the road block. There is another blank line between the instructions to `mark` and the instructions to `ann`.
- Line up curly braces so that the closing brace is directly beneath the opening brace.
- Indent everything inside a pair of braces by a consistent number of spaces (this book uses two).

Many conventions govern indenting programs and lining up braces. None of them are right or wrong, but are subject to personal preference. Some people like the preceding style shown through this textbook because it is easy to ensure that braces match.

Nevertheless, your instructor or future employer may have other conventions, complete with reasons to support their preference. As an employee (or student), you will need to accommodate their preferences.

Programs are available that can reformat code to make it adhere to a specific set of conventions. This book's Web site contains references to at least one such program. It is often possible to configure your IDE so that using a code reformatter is as easy as clicking a button.

2.4.2 Identifiers

The symbols that make up a Java program are divided into three groups: special symbols, reserved words, and identifiers. **Special symbols** include the braces `{` and `}`, periods, semicolons, and parentheses. **Reserved words**, also called **keywords**, have a special meaning to the compiler. They include `class`, `package`, `import`, `public`, and `int`. A complete list of reserved words is shown in Table 2-2. Finally, **identifiers** are names: the names of variables (`karel`), the names of classes (`Robot`), the names of services (`move`), and the names of packages (`becker.robots`).

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

(table 2-2)

Java reserved words. `const` *and* `goto` *are reserved but not currently used*

Programmers have lots of choice in the names they use as identifiers. Wise choices make a program much easier to read and understand, thereby increasing the likelihood that it is correct. Programs with well-chosen identifiers are self-documenting and need fewer explanatory comments.

A well-chosen identifier clearly indicates the purpose of the thing it names. It is better to name a class `Robot` than `R`, for instance. Naming an icon `onIcon` is much better than `on` or `icon` or even just `i`.

KEY IDEA

A name should clearly reveal the purpose of what it names.

Balanced with the need for a clear intent are readability and brevity for the sake of the person who must type the name over and over. Naming a robot `robotThatMovesTheThingFrom2_1To3_2` is clearly overkill.

When naming parts of their program, most Java programmers use conventions established by Java's creators. The name of a class, for example, should be a descriptive, singular noun such as `Robot`, `Wall`, or `Lamp`. Class names begin with an uppercase letter followed by lowercase letters. If the name is composed of two or more words, such as `CircleIcon`, then capitalize each word.

KEY IDEA

Naming conventions make it easier to recognize what an identifier represents.

The name of a variable should be a descriptive noun or noun phrase: `warningLamp`, `westWall`, or a robot named `collectorRobot` in a program where lots of things are collected. The variable name should describe what the variable represents. A variable name begins with a lowercase letter. Names composed of more than one word should have the first letter of each subsequent word capitalized, as in `collectorRobot`. Variable names can also contain digits and underscores after the first letter, but not spaces, tabs, or punctuation.

A method name should be a verb or verb phrase that describes what the method does. Like a variable name, a method name begins with a lowercase letter. In names composed of two or more words, the subsequent words are capitalized. Examples we have seen so far include `move`, `turnLeft`, `turnAround`, and `setLocation`.

Capitalization matters in identifiers. `westWall` is not the same as `westwall`. The Java compiler notices the difference between the uppercase `W` and the lowercase `w`—even if we don't—and treats them as two different identifiers. Table 2-3 summarizes the Java naming conventions.

(table 2-3)

Summary of naming conventions

Identifier	Conventions	Examples

Class	A descriptive singular noun, beginning with an uppercase letter. If the name is composed of several words, each word begins with a capital letter.	`Robot`
		`Lamp`
		`CircleIcon`

Method	A descriptive verb or verb phrase, beginning with a lowercase letter. If the method name is composed of several words, each word, except the first, begins with a capital letter.	`move`
		`pickThing`
		`canPickThing`
		`setSize`

Variable	A descriptive noun or noun phrase, beginning with a lowercase letter. If the name is composed of several words, each word, except the first, begins with a capital letter.	`karel` `newYorkCity`

2.4.3 Comments

KEY IDEA

Comments do not affect the execution of the program.

Comments are annotations inserted by the programmer to help others understand how to use the code she is writing, how it works, or how to modify it. The comments help establish the context of a statement or block of code so that readers can more quickly understand the code. Comments are for people; they do not affect the execution of the program. In this way, they are like white space.

An excellent practice is to first write a comment that states what this section of your program must do. Then write the code to implement your comment. Clearly state in English what you are trying to do, then explain how in Java. This two-part practice helps keep things clear in your mind, minimizing errors and speeding debugging.

Java has three different kinds of comments: single-line comments, multi-line comments, and documentation comments.

Single-Line Comments

A **single-line comment** begins with two consecutive slashes and extends to the end of the line. Single-line comments have already been used in the programs in Chapter 1 to document the purpose of a block of code. The first line in the following block of code is a single-line comment:

```
// Use the robot to move one of the lamps.
lampMover.move();
lampMover.pickThing();
lampMover.move();
lampMover.putThing();
lampMover.move();
```

The comment explains the intent of the code; it does not repeat how the code works. A reader may then consider whether the intent is appropriate at this point in the program, and whether the code correctly carries out the intent.

It is also possible to put a single-line comment at the end of a line of code, as shown in the following line from the `FramePlay` program in Chapter 1:

```
import javax.swing.*;     // use JFrame, JPanel, JButton, JTextArea
```

Multi-Line Comments

If you have more to say than will fit on a single line consider using a multi-line comment. A **multi-line comment** begins with `/*` and extends, possibly over many lines, until the next `*/`. The following example is a multi-line comment extending over three lines.

```
/* Set up the initial situation to match the figure given for problem 5.
   It consists of eight walls positioned to form a 2x2 square.
*/
```

Such a comment should go immediately before the first line of code that implements what is described.

Another use of a multi-line comment is to temporarily remove some lines of code, perhaps so another approach can be tested without losing previous work. For example, in Section 2.2.6, we explored two ways to implement `turnRight`. The programmer could have started with the solution that uses `turnLeft` three times. Perhaps she then thought of the other solution, which turns around first. If she wasn't quite sure the second solution would work, her code might have looked like this:

```
public void turnRight()
{ /*
   this.turnLeft();
   this.turnLeft();
   this.turnLeft();
   */
   this.turnAround();
   this.turnLeft();
}
```

KEY IDEA

Use single-line comments to annotate code and multi-line comments to comment out existing code.

This practice is called **commenting out code**. One of the consequences of using the next `*/` to end a comment is that a multi-line comment can not include another (nested) multi-line comment. That is, you can't comment out multi-line comments. For this reason, some programmers reserve multi-line comments for commenting out code, using single-line comments for actual annotations.

Documentation Comments

Single-line and multi-line comments explain the intent of the code for people reading the code. **Documentation comments** are used to generate Web-based documentation like that shown in Figure 1-15 through Figure 1-18. They describe the purpose of a class, constructor, or method and provide information useful to people who want to use them without understanding how they work internally.

KEY IDEA

Document the purpose of each class, constructor, and method.

Each class, constructor, and method should have a documentation comment. The comment must appear immediately before the class, method, or constructor is declared; that is, immediately before the line containing `public class «className» extends «superClass»` or `public void «methodName»()`.

A documentation comment is similar to a multi-line comment, except that it begins with `/**` rather than `/*`. Another difference is that a documentation comment may contain **tags** to identify specific information that should be displayed distinctively on a Web page. For example, the documentation comment for a class may contain the `@author` tag to identify the person who wrote the class.

One of the most important tags is `@param`. It allows you to document each parameter's purpose. The `@param` tag is followed by the name of the parameter and a description of that parameter.

Listing 2-10 shows the `ExperimentRobot` listing again, this time with appropriate documentation comments.

Listing 2-10: *A listing of* `ExperimentRobot` *showing appropriate documentation*

```
import becker.robots.*;

/** A new kind of robot that can turn around, turn right, and move forward three intersections
 *   at a time.
 *   @author Byron Weber Becker  */
public class ExperimentRobot extends Robot
{
   /** Construct a new ExperimentRobot.
    *   @param aCity          The city in which the robot will be located.
    *   @param aStreet        The robot's initial street.
```

Listing 2-10: *A listing of* `ExperimentRobot` *showing appropriate documentation* (continued)

```
     *  @param anAvenue        The robot's initial avenue.
     *  @param aDirection      The robot's initial direction. */
    public ExperimentRobot(City aCity, int aStreet,
                           int anAvenue, Direction aDirection)
    { super(aCity, aStreet, anAvenue, aDirection);
    }

    /** Turn the robot around so it faces the opposite direction. */
    public void turnAround()
    { this.turnLeft();
      this.turnLeft();
    }

    /** Move the robot forward three times. */
    public void move3()
    { this.move();
      this.move();
      this.move();
    }

    /** Turn the robot right 90 degrees by turning left. */
    public void turnRight()
    { this.turnAround();
      this.turnLeft();
    }
  }
```

2.4.4 External Documentation (advanced)

The tool that extracts the documentation comments and formats them for the Web is known as `javadoc`. Your development environment might have a simpler way to run this tool; if it does not, you can perform the following steps on a computer running Windows (with analogous steps on other computers).

- Open a window for a command-prompt.
- Change directories to the directory containing the Java source code.
- Run `javadoc`, specifying the Java files to process and where the output should be placed.
- View the result with a Web browser.

For example, suppose we want to generate documentation for the three classes used in the experiment at the beginning of the chapter and that the files reside in the directory `E:\experiment`. In the command-prompt window, enter the following commands (the > is called a **prompt** and is displayed by the system to indicate that it is ready to accept your input).

```
> E:
> cd experiment
> javadoc -d doc -classpath e:/robots/becker.jar *.java
```

The first two commands change the focus of the command prompt, first to the correct disk drive (`E:`), and then to the correct directory on that disk (`experiment`). The last command starts the program that produces the Web pages. This `javadoc` command has three sets of parameters:

- `-d doc` specifies that the documentation should be placed in a directory named `doc`.
- `-classpath e:/robots/becker.jar` indicates where to find the `becker` library. It allows `javadoc` to include relevant information about robot classes. You will need to replace the path given here with the location of the library on your computer.
- `*.java` means that `javadoc` should process all the files in the current directory that end with `.java`.

Depending on how your software is installed, the system may not find the `javadoc` program. In that case, you need to find `javadoc`'s location on the disk drive and provide the complete path to it. For example, on my system, I used the search feature of the Windows Explorer program to search for `javadoc`. The results told me that the program was in `C:\java\jdk1.5\bin\javadoc.exe`. I could then modify the last line as follows:

```
> C:\java\jdk1.5\bin\javadoc -d doc -classpath e:/robots/becker.jar *.java
```

Alternatively, you may be able to set your system's `path` variable to include the directory containing `javadoc`.

2.5 Meaning and Correctness

Nothing prevents a programmer from implementing a method called `move3` with 5, 10, or even 100 calls to `move`. In fact, nothing requires a method named `move3` to even contain `move` commands. It could be defined as follows:

PATTERN

Parameterless Command

```
public void move3()
{ this.turnLeft();
  this.pickThing();
  this.turnLeft();
}
```

If we defined the `move3` method this way, someone else reading our program would be confused and surprised. Over time, we could confuse ourselves, introducing errors into the program in spite of defining `move3` ourselves to behave in this manner.

The meaning of a command is the list of commands contained in its body, not its name. When a program is executed, the command does exactly what the commands in the body instruct it to do. There is no room for interpretation.

KEY IDEA

Use meaningful names.

A good programmer gives each command a meaningful name. Another person should be able to make a reasonable guess about what the command does from its name. There should be no surprises such as the robot picking something up in the middle of a command whose name does not imply picking things up.

When we write new programs, it is common to trace the program by hand to verify how it behaves. Because the command name only implies what it does, it is important to trace the actual instructions in each command. The computer cannot and does not interpret the names of commands when it executes a program; we shouldn't either when we trace a program manually.

KEY IDEA

Correct methods meet their specifications.

The correctness of a command is determined by whether it fulfills its **specification**. The specification is a description of what the method is supposed to do. The specification might be included in the method's documentation or in the problem statement given by your instructor. A command may be poorly named, but still correct. For instance, the specification of `ExperimentRobot` at the beginning of the chapter requires a command to turn the robot around. It could have been given the idiotic name of `doIt`. As long as `doIt` does, indeed, turn the robot around (and nothing else) the specification is met and the command is correct.

Because the `move3` command is simple, it is easy to convince ourselves that it is correct. Many other commands are much more complex, however. The correctness of these commands must be verified by writing test programs that execute the command, checking the actual result against the expected result. This practice is not foolproof, however. Conditions in which the command fails may not be tested and go undetected.

A correct command, such as `move3`, can be used incorrectly. For example, a client can place an `ExperimentRobot` facing a wall. Instructing this robot to `move3` will result in an error when the robot attempts to move into the wall. In this case, we say the command's **preconditions** have not been met. Preconditions are conditions that must be true for a command to execute correctly.

2.6 Modifying Inherited Methods

KEY IDEA

A subclass can make limited modifications to inherited methods as well as add new methods.

Besides adding new services to an object, sometimes we want to modify existing services so that they do something different. We might use this facility to make a dancing robot that, when sent a `move` message, first spins around on its current intersection and then moves. We might build a kind of robot that turns very fast even though it continues to move relatively slowly, or (eventually), a robot that checks to see if something is present before attempting to pick it up. In a graphical user interface, we might make a special kind of component that paints a picture on itself. In all of these situations, we replace the definition of a method in a superclass with a new definition. This replacement process is called **overriding**.

2.6.1 Overriding a Method Definition

To override the definition of a method, you create a new method with the same name, return type, and parameters in a subclass. These constitute the method's **signature**. As an example, let's create a new kind of robot that can turn left quickly. That is, we will override the `turnLeft` method with a new method that performs the same service differently.

You may have noticed that the online documentation for `Robot` includes a method named `setSpeed`, which allows a robot's speed to be changed. Our general strategy will be to write a method that increases the robot's speed, turns, and then returns the speed to normal. Turning quickly doesn't seem to be something we would use often, so it has not been added to `RobotSE`. On the other hand, it seems reasonable that fast-turning robots need to turn around and turn right, so our new class will extend `RobotSE`.

As the first step in creating the `FastTurnBot` class, we create the constructor and the shell of the new `turnLeft` method, as shown in Listing 2-11.

FIND THE CODE

ch02/override/

PATTERN

Extended Class

Listing 2-11: *An incomplete class which overrides* `turnLeft`

```
import becker.robots.*;

/** A FastTurnBot turns left very quickly relative to its normal speed.
 *   @author Byron Weber Becker  */
public class FastTurnBot extends RobotSE
{
   /** Construct a new FastTurnBot.
    * @param aCity          The city in which the robot appears.
    * @param aStreet        The street on which the robot appears.
    * @param anAvenue       The avenue on which the robot appears.
    * @param aDirection     The direction the robot initially faces. */
```

Listing 2-11: *An incomplete class which overrides* `turnLeft` (continued)

```
   public FastTurnBot(City aCity, int aStreet, int anAvenue,
                              Direction aDirection)
   { super(aCity, aStreet, anAvenue, aDirection);
   }

   /** Turn 90 degrees to the left, but do it more quickly than normal. */
   public void turnLeft()
   {
   }
}
```

PATTERN

Constructor

When this class is instantiated and sent a `turnLeft` message, it does nothing. When the message is received, Java starts with the object's class (`FastTurnBot`) and looks for a method matching the message. It finds one and executes it. Because the body of the method is empty, the robot does nothing.

How can we get it to turn again? We *cannot* write `this.turnLeft();` in the body of `turnLeft`. When a `turnLeft` message is received, Java finds the `turnLeft` method and executes it. The `turnLeft` method then executes `this.turnLeft`, sending *another* `turnLeft` message to the object. Java finds the same `turnLeft` method and executes it. The process of executing it sends *another* `turnLeft` message to the object, so Java finds the `turnLeft` method again, and repeats the sequence. The program continues sending `turnLeft` messages to itself until it runs out of memory and crashes. This problem is called **infinite recursion**.

LOOKING AHEAD

Recursion occurs when a method calls itself. Although recursion causes problems in this case, it is a powerful technique.

What we really want is the `turnLeft` message in the `FastTurnBot` class to execute the `turnLeft` method in a superclass. We want to send a `turnLeft` message in such a way that Java begins searching for the method in the superclass rather than the object's class. We can do so by using the keyword `super` instead of the keyword `this`. That is, the new definition of `turnLeft` should be as follows:

KEY IDEA

Using `super` *instead of* `this` *causes Java to search for a method in the superclass rather than the object's class.*

```
public void turnLeft()
{ super.turnLeft();
}
```

We have returned to where we started. We have a robot that turns left at the normal speed. When a `FastTurnBot` is sent a `turnLeft` message, Java finds this `turnLeft` method and executes it. This method sends a message to the superclass to execute its `turnLeft` method, which occurs at the normal speed.

To make the robot turn faster, we add two calls to `setSpeed`, one before the call to `super.turnLeft()` to increase the speed, and one more after the call to decrease the speed back to normal. The documentation indicates that `setSpeed` requires a single parameter, the number of moves or turns the robot should make in one second.

The default speed of a robot is two moves or turns per second. The following method uses `setSpeed` so the robot turns 10 times as fast as normal, and then returns to the usual speed.

```
public void turnLeft()
{ this.setSpeed(20);
  super.turnLeft();
  this.setSpeed(2);
}
```

The `FastTurnBot` class could be tested with a small program such as the one in Listing 2-12. Running the program shows that `speedy` does, indeed, turn quickly when compared to a `move`.

FIND THE CODE

ch02/override/

Listing 2-12: *A program to test a* `FastTurnBot`

```
import becker.robots.*;

/** A program to test a FastTurnBot.
 *  @author Byron Weber Becker  */
public class Main extends Object
{
  public static void main(String[] args)
  { City cairo = new City();
    FastTurnBot speedy = new FastTurnBot(
                    cairo, 1, 1, Direction.EAST);

    speedy.turnLeft();
    speedy.move();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.move();
  }
}
```

2.6.2 Method Resolution

So far, we have glossed over how Java finds the method to execute, a process called **method resolution**. Consider Figure 2-13, which shows the class diagram of a `FastTurnBot`. Details not relevant to the discussion have been omitted, including constructors, attributes, some services, and even some of the `Robot`'s superclasses (represented by an empty rectangle). The class named `Object` is the superclass, either directly or indirectly, of every other class.

When a message is sent to an object, Java always begins with the object's class, looking for a method implementing the message. It keeps going up the hierarchy until it either finds a method or it reaches the ultimate superclass, `Object`. If it reaches `Object` without finding an appropriate method, a compile-time error is given.

Let's look at several different examples. Consider the following code:

```
FastTurnBot speedy = new FastTurnBot(...);
speedy.move();
```

To execute the `move` method, Java begins with `speedy`'s class, `FastTurnBot`, in the search for the method. When Java doesn't find a method named `move` in `FastTurnBot`, it looks in `RobotSE` and then in `Robot`, where a method matching the `move` method is found and executed.

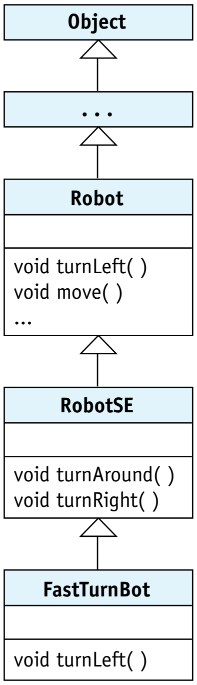
As another example, consider the following code:

```
RobotSE special = new RobotSE(...);
special.move();
```

The search for a `move` method begins with `RobotSE`, the class that instantiated `special`. It doesn't matter that `RobotSE` has been extended by another class; what matters is that when `special` was constructed, the programmer used the constructor for `RobotSE`. Therefore, searches for methods begin with `RobotSE`.

(figure 2-13)

Class diagram of a `FastTurnBot`



Once again, consider `speedy`. What happens if `speedy` is sent the `turnAround` message? The search for the `turnAround` method begins with `speedy`'s class, `FastTurnBot`. It's found in `RobotSE` and executed. As it is executed, it calls `turnLeft`. Which `turnLeft` method is executed, the one in `FastTurnBot` or the one in `Robot`?

KEY IDEA

Overriding a method can affect other methods that call it, even methods in a superclass.

The `turnLeft` message in `turnAround` is sent to the implicit parameter, `this`. The implicit parameter is the same as the object that was originally sent the message, `speedy`. So Java begins with `speedy`'s class, searching for `turnLeft`. It finds the method that turns quickly and executes it. Therefore, a subclass can affect how methods in a superclass are executed.

LOOKING AHEAD

Written Exercise 2.4 asks you to trace similar examples.

If `turnAround` is written as follows, the result would be different.

```
public void turnAround()
{ super.turnLeft();
  super.turnLeft();
}
```

KEY IDEA

The search for the method matching a message sent to `super` *begins in the method's superclass.*

Now the search for `turnLeft` begins with the superclass of the class containing the method, or `Robot`. `Robot` contains a `turnLeft` method. It is executed, and the robot turns around at the normal pace.

Suppose you occasionally want `speedy` to turn left at its normal speed. Can you somehow skip over the new definition of `turnLeft` and execute the normal one, the one that was overridden? No. If we really want to execute the original `turnLeft`, we should not have overridden it. Instead, we should have simply created a new method, perhaps called `fastTurnLeft`.

2.6.3 Side Effects

`FastTurnBot` has a problem, however. Suppose that Listing 2-12 contained the statement `speedy.setSpeed(20);` just before line 12. This statement would speed `speedy` up dramatically. Presumably, the programmer wanted `speedy` to be speedier than normal all of the time. After its first `turnLeft`, however, `speedy` would return to its normal pace of 2 moves per second.

This phenomenon is called a **side effect**. Invoking `turnLeft` changed something it should not have changed. Our programmer will be very annoyed if she must reset the speed after every command that turns the robot. Ideally, a `FastTurnBot` returns to its previous speed after each turn.

KEY IDEA

Not only are side effects annoying, they can lead to errors. Avoid them where possible; otherwise, document them.

The programmer can use the `getSpeed` query to find out how long the robot currently takes to turn. This information can be used to adjust the speed to its original value after the turn is completed. The new version of `turnLeft` should perform the following steps:

```
set the speed to 10 times the current speed
turn left
set the speed to one-tenth of the (now faster) speed
```

The query `this.getSpeed()` obtains the current speed. Multiplying the speed by 10 and using the result as the value to `setSpeed` increases the speed by a factor of 10. After the turn, we can do the reverse to decrease the speed to its previous value, as shown in the following implementation of `turnLeft`:

LOOKING AHEAD

Another approach is to remember the current speed. When the robot is finished turning, set the speed to the remembered value. More in Chapters 5 and 6.

```
public void turnLeft()
{ this.setSpeed(this.getSpeed() * 10);
  super.turnLeft();
  this.setSpeed(this.getSpeed() / 10);
}
```

Using queries and doing arithmetic will be discussed in much more detail in the following chapters.

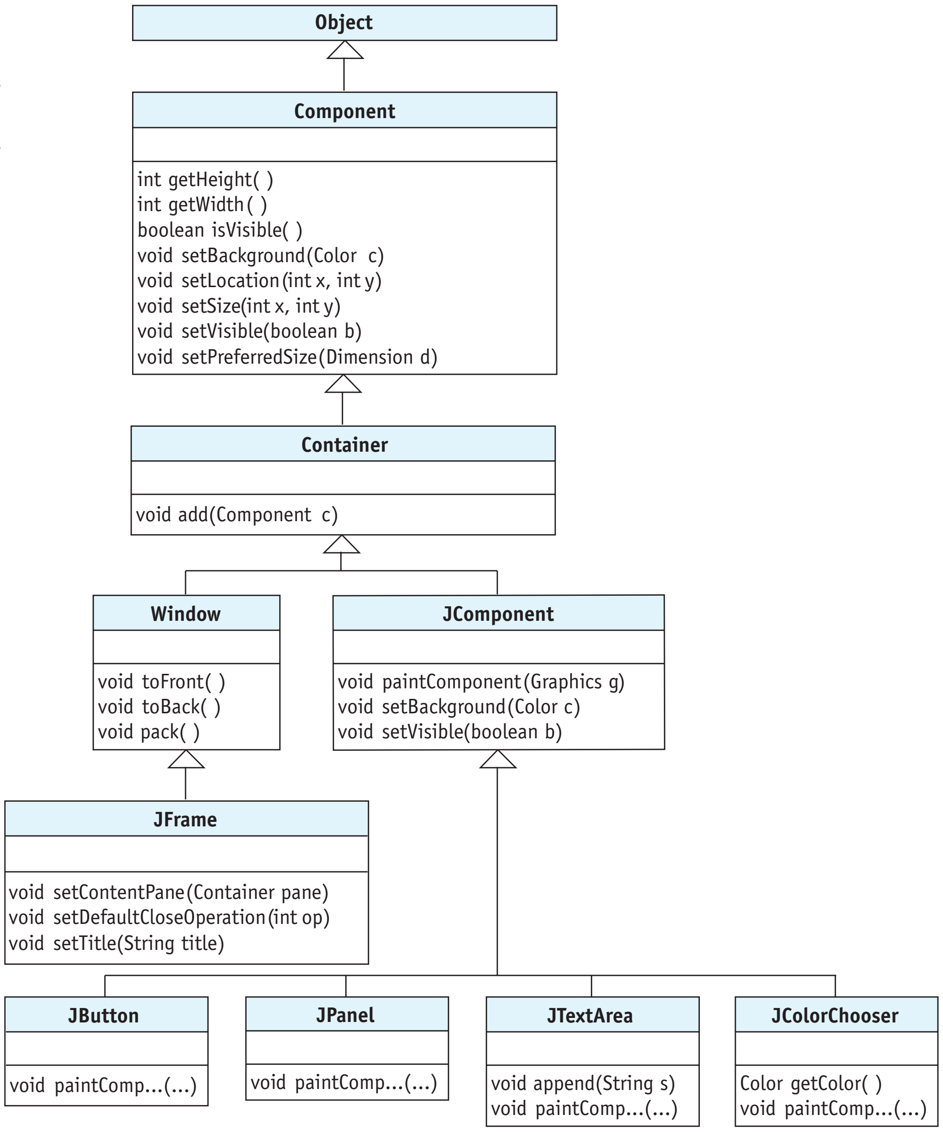
2.7 GUI: Extending GUI Components

The Java package that implements user interfaces is known as the **Abstract Windowing Toolkit** or **AWT**. A newer addition to the AWT is known as **Swing**. These packages contain classes to display components such as windows, buttons, and textboxes. Other classes work to receive input from the mouse, to define colors, and so on.

The AWT and Swing packages make extensive use of inheritance. Figure 2-14 contains a class diagram showing a simplified version of the inheritance hierarchy. Many classes are omitted, as are many methods and all attributes.

(figure 2-14)

Simplified class diagram showing the inheritance hierarchy for some `AWT` *and* `Swing` *classes*


Object
Component
int getHeight( )
int getWidth( )
boolean isVisible( )
void setBackground(Color c)
void setLocation(int x, int y)
void setSize(int x, int y)
void setVisible(boolean b)
void setPreferredSize(Dimension d)
Container
void add(Component c)
Window
void toFront( )
void toBack( )
void pack( )
JComponent
void paintComponent(Graphics g)
void setBackground(Color c)
void setVisible(boolean b)
JFrame
void setContentPane(Container pane)
void setDefaultCloseOperation(int op)
void setTitle(String title)
JButton
void paintComp...(...)
JPanel
void paintComp...(...)
JTextArea
void append(String s)
void paintComp...(...)
JColorChooser
Color getColor( )
void paintComp...(...)

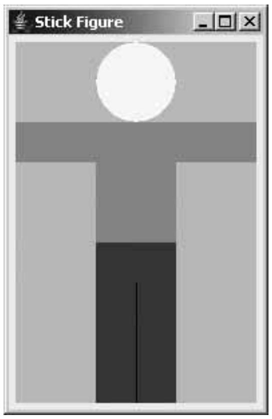
One new aspect of this class diagram is that some classes have two or more subclasses. For example, `Container` is the superclass for both `Window` and `JComponent`. The effect is that `Window` objects and `JComponent` objects (and their subclasses) have much in common—the set of services they inherit from the `Container` and `Component` classes.

The class diagram reveals several new pieces of information:

- When we implemented the `FramePlay` program in Listing 1-6, we sent six different messages to a `JFrame` object: `setContentPane`, `setTitle`, `setDefaultCloseOperation`, `setLocation`, `setSize`, and `setVisible`. We now realize that only three of these are actually declared by the `JFrame` class. The other three services are offered by `JFrame` because they are inherited from `Component`.
- Because `JFrame`, `JPanel`, `JButton`, and so on, all indirectly extend `Component`, they can all answer queries about their width, height, and visibility, and can all[2] set their background color, position, size, and visibility.
- The `JComponent` class overrides two of the services provided by `Component`. `JComponent` must be doing something extra for each of those services.
- The statement `contents.add(saveButton);` in the `FramePlay` program added a button to an instance of `JPanel`. We now see that `add` is actually a service of the `Container` class, inherited by `JPanel`.
- Each of the classes extending `JComponent` inherits the method `paintComponent`. Perhaps if this method were overridden, we could affect how the component looks. This result would, indeed, be the case and is the topic of the next section.

2.7.1 Extending `JComponent`

In this section we will write a program that paints a picture. Figure 2-15 shows a simple stick figure. When viewed in color, the pants are blue, the shirt is red, and the head is yellow.



(figure 2-15)

Simple stick figure

[2] There is, unfortunately, some fine print. The statements above are true, but in some circumstances you can't see the results. For example, setting the background color of a `JFrame` doesn't appear to have an effect because the content pane completely covers the `JFrame`, and you see the content pane's color.

Our strategy is to create a new class, `StickFigure`, which extends `JComponent`. We choose to extend `JComponent` because it is the simplest of the components shown in the class diagram, and it doesn't already have its own appearance. We will extend it by overriding `paintComponent`, the method responsible for the appearance of the component. As we did with the several components in the `FramePlay` program in Listing 1-6, the stick figure component will be placed in a `JPanel`. The `JPanel` will be set as the content pane in a `JFrame`.

Listing 2-13 shows the beginnings of the `StickFigure` class. It provides a parameterless constructor and nothing more. The constructor doesn't need parameters because `JComponent` has a constructor that does not need parameters. Our constructor calls `JComponent`'s constructor by invoking `super` without parameters.

The constructor performs one important task: in lines 13–14 it specifies a preferred size for the stick figure component. The preferred size says how many pixels wide and high the component should be, if possible. Line 13 creates a `Dimension` object 180 pixels wide and 270 pixels high. The next line uses this object to set the preferred size for the stick figure.

FIND THE CODE

ch02/stickFigure/

PATTERN

Extended Class Constructor

Listing 2-13: *An extended* `JComponent`

```
 1 import javax.swing.*;      // JComponent
 2 import java.awt.*;         // Dimension
 3
 4 /** A new kind of component that displays a stick figure.
 5 *
 6 *   @author Byron Weber Becker */
 7 public class StickFigure extends JComponent
 8 {
 9   public StickFigure()
10   { super();
11
12     // Specify the preferred size for this component
13     Dimension prefSize = new Dimension(180, 270);
14     this.setPreferredSize(prefSize);
15   }
16 }
```

It is also possible to reduce lines 13 and 14 down to a single line:

```
this.setPreferredSize(new Dimension(180, 270));
```

This creates the object and passes it to `setPreferredSize` without declaring a variable. We can avoid declaring the variable if we don't need to refer to the object in the future (as with `Wall` and `Thing` objects), or we can pass it to the only method that requires it as soon as it's created, as we do here.

KEY IDEA

Sometimes we don't need a variable to store object references.

Now would be a good time to implement the `main` method for the program. By compiling and running the program early in the development cycle, we can often catch errors in our thinking that may be much more difficult to change later on. Listing 2-14 shows a program for this purpose. Running it results in an empty frame as shown in Figure 2-16. It follows the Display a Frame pattern and consequently it is similar to the `FramePlay` program in Listing 1-6.

FIND THE CODE

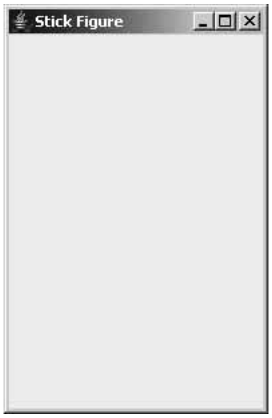
ch02/stickFigure/

Listing 2-14: *A program that uses a class extending* `JComponent`

```
 1  import javax.swing.*;
 2
 3  /** Create a stick figure and display it in a frame.
 4   *
 5   *  @author Byron Weber Becker */
 6  public class Main
 7  {
 8    public static void main(String[ ] args)
 9    { // Declare the objects to show.
10      JFrame frame = new JFrame();
11      JPanel contents = new JPanel();
12      StickFigure stickFig = new StickFigure();
13
14      // Add the stick figure to the contents.
15      contents.add(stickFig);
16
17      // Display the contents in a frame.
18      frame.setContentPane(contents);
19      frame.setTitle("Stick Figure");
20      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21      frame.setLocation(250, 100);
22      frame.pack();
23      frame.setVisible(true);
24    }
25  }
```

One difference between this program and the `FramePlay` program and the pattern is in how the frame is sized. The previous program explicitly set the size of the frame using the `setSize` method. This version uses the method `pack` in line 22. This method uses the preferred sizes of all the components to calculate the best size for the frame.

The result of running this program is shown in Figure 2-16. It looks exactly like an empty `JFrame` because the `JComponent` is invisible until we override `paintComponent` to change its appearance.



(figure 2-16)

Result of running the program in Listing 2-14 with the incomplete `StickFigure` *class from Listing 2-13*

2.7.2 Overriding `paintComponent`

To actually draw the stick figure, we need to override `paintComponent` to provide it with additional functionality. We know from both the class diagram in Figure 2-14 and the online documentation that `paintComponent` has a parameter of type `Graphics`. This parameter is often named simply `g`. We will have much more to say about parameters in later chapters. For now, we will just say that `g` is a reference to an object that is used for drawing. It is provided by the client that calls `paintComponent` and may be used by the code contained in the `paintComponent` method.

LOOKING AHEAD

We are practicing incremental development: code a little, test a little, code a little, test a little. For more on development strategies, see Chapter 11.

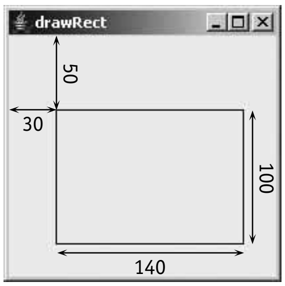
The superclass's implementation of `paintComponent` may have important work to do, and so it should be called with `super.paintComponent(g)`. It requires a `Graphics` object as an argument, and so we pass it `g`, the `Graphics` object received as a parameter. Doing so results in the following method. The method still has not added any functionality, but adding it to Listing 2-13 between lines 14 and 15 still results in a running program.

```
public void paintComponent(Graphics g)
{ super.paintComponent(g);
}
```

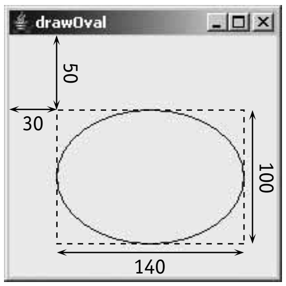
The `Graphics` parameter, `g`, provides services such as `drawRect`, `drawOval`, `drawLine`, and `drawString`, each of which draw the shape described in the service's name. A companion set of services includes `fillRect` and `fillOval`, each of which also draws the described shape and then fills the interior with a color. The color used is determined by the most recent `setColor` message sent to `g`. The color specified is used until the next `setColor` message.

All of the `draw` and `fill` methods require parameters specifying where the shape is to be drawn and how large it should be. Like positioning a frame, measurements are given in relation to an origin in the upper-left corner, and are in pixels.

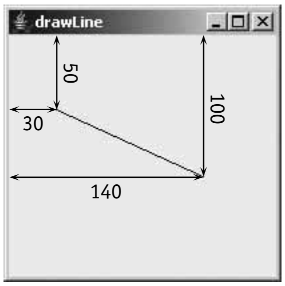
Figure 2-17 shows the relationship between the parameters and the figure that is drawn. For `drawRect` and `drawOval`, the first two parameters specify the position of the upper left corner of the figure, while the third and fourth parameters specify the width and height. For an oval, the width and height are of the smallest box that can contain the oval. This box is called the **bounding box** and is shown in Figure 2-17 as a dashed line. Of course, the bounding box is not actually drawn on the screen.



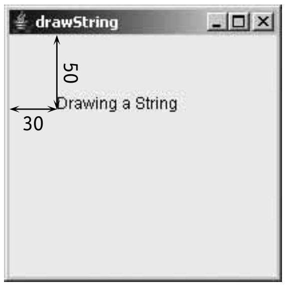
```
g.drawRect(30, 50, 140, 100);
```



```
g.drawOval(30, 50, 140, 100);
```



```
g.drawLine(30, 50, 140, 100);
```



```
g.drawString("Drawing a String", 30, 50);
```

(figure 2-17)

Relationship between the arguments and the effects of four drawing methods

In each of these methods, the order of the arguments is *x* before *y* and *width* before *height*.

The parameters for a line are different from the parameters for rectangles and ovals. The first two parameters specify one end of the line in relation to the origin, while the last two parameters specify the other end of the line in relation to the origin.

The `drawString` method takes a string as the first parameter and the position of the first letter as the second and third parameters.

With this background information, we can finally add the statements to draw the stick figure. The complete code for the `stickFigure` class is given in Listing 2-15. Running it with the `main` method in Listing 2-14 produces the image shown in Figure 2-15.

FIND THE CODE

ch02/stickFigure/

PATTERN

Constructor

PATTERN

Parameterless Command

Listing 2-15: *Overriding* `paintComponent` *to draw a stick figure*

```
import javax.swing.*;      // JComponent
import java.awt.*;         // Dimension

/** A new kind of component that displays a stick figure.
 *
 *  @author Byron Weber Becker */
public class StickFigure extends JComponent
{
  public StickFigure()
  { super ();
    Dimension prefSize = new Dimension(180, 270);
    this.setPreferredSize(prefSize);
  }

  // Paint a stick figure.
  public void paintComponent(Graphics g)
  { super.paintComponent(g);

    // Paint the head.
    g.setColor(Color.YELLOW);
    g.fillOval(60, 0, 60, 60);

    // Paint the shirt.
    g.setColor(Color.RED);
    g.fillRect(0, 60, 180, 30);
    g.fillRect(60, 60, 60, 90);

    // Paint the pants.
    g.setColor(Color.BLUE);
    g.fillRect(60, 150, 60, 120);
    g.setColor(Color.BLACK);
    g.drawLine(90, 180, 90, 270);
  }
}
```

2.7.3 How `paintComponent` Is Invoked

You may have noticed that the `paintComponent` method is *not* called from anywhere in Listing 2-15 or the client code shown in Listing 2-14. Look all through the code, and you will not find an instance of the Command Invocation pattern `stickFig.paintComponent(g);`. Yet we know it is invoked because it paints the stick figure. How?

In the Sequential Execution pattern in Chapter 1, we described statements as being executed one after another, as if they were strung on a thread of string. A computer program can have two or more of these **threads**, each with their own sequence of statements. The program we just wrote has at least two threads. The first one is in the `main` method. It creates a `JFrame` and invokes a number of its commands such as `setDefaultCloseOperation` and `setVisible`. When it gets to the end of the `main` method, that thread ends.

When a `JFrame` is instantiated, a second thread begins. This is *not* a normal occurrence when an object is instantiated; `JFrame`'s authors deliberately set up the new thread. `JFrame`'s thread monitors the frame and detects when it has been damaged and must be repainted. A frame can be damaged in many ways. It is damaged when the user resizes it by dragging a border or clicking the minimize or maximize buttons. It's damaged when it is first created because it hasn't been drawn yet. It's also damaged if another window is placed on top of it and then moved again. In each of these cases, the second thread of control calls `paintComponent`, providing the `Graphics` object that `paintComponent` should draw upon.

LOOKING AHEAD

We will use this capability in Section 3.5.2 to make two or more robots move simultaneously.

KEY IDEA

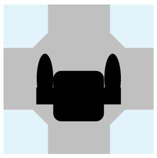
`paintComponent` *is called by "the system." We don't call it.*

2.7.4 Extending `Icon`

We learned in Section 2.3.1 that `Icon` is the class used to represent images of things in the robot world—robots, intersections, things, flashers, walls, and so on—all use icons to display themselves. As you might expect, `Icon` has been extended a number of times to provide different icons for different kinds of things. The documentation references classes named `FlasherIcon`, `RobotIcon`, `WallIcon`, and so on.

You, too, can extend the `Icon` class to create your own custom icons. The example shown in Figure 2-18 was produced by the code shown in Listing 2-16.

As with any other subclass, it gives the name of the class it extends (line 5). Before that are the packages it relies on. In this case, it imports the `Icon` class from the `becker.robots.icons` package and the `Graphics` class from `java.awt`.



(figure 2-18)

Custom robot icon

One difference, when compared to extending `JComponent`, is that we must override a method named `paintIcon` instead of `paintComponent`. This fact can be gleaned from reading the documentation for `Icon`. Like `paintComponent`, `paintIcon` has a parameter of type `Graphics` to use for the actual drawing.

An icon is always painted in a standard 100 × 100 pixel space facing north. Lines 14–23 in Listing 2-16 draw the robot in this position. Other parts of the robot system scale and rotate the icons, as necessary.

FIND THE CODE

ch02/extendIcon/

Listing 2-16: *Code for a customized robot icon*

```
 1  import becker.robots.icons.*;      // Icon
 2  import java.awt.*;                 // Graphics, Color
 3
 4  /** Create a robot icon that has arms. */
 5  public class ArmRobotIcon extends Icon
 6  {
 7     /** Create a new icon for a robot. */
 8     public ArmRobotIcon()
 9     { super();
10     }
11
12     /** Paint the icon. */
13     public void paintIcon(Graphics g)
14     { g.setColor(Color.BLACK);
15
16        // body
17        g.fillRoundRect(35, 35, 30, 30, 10, 10);
18        // shoulders
19        g.fillRect(25, 45, 10, 10);
20        g.fillRect(65, 45, 10, 10);
21        // arms
22        g.fillOval(25, 25, 10, 30);
23        g.fillOval(65, 25, 10, 30);
24     }
25  }
```

Use the `setIcon` method to change the icon used to display a robot. One way to call `setIcon` is to create a new class of robots, as shown in Listing 2-17.

Listing 2-17: *An `ArmRobot` uses an `ArmRobotIcon` to display itself*

FIND THE CODE

ch02/extendIcon/

```
import becker.robots.*;

/** A robot with an icon that shows arms. */
public class ArmRobot extends Robot
{
   /** Construct a new ArmRobot.
    *  @param aCity          The City where the robot will reside.
    *  @param aStreet        The robot's initial street.
    *  @param anAvenue       The robot's initial avenue.
    *  @param aDirection     The robot's initial direction. */
   public ArmRobot(City aCity, int aStreet, int anAvenue,
                           Direction aDirection)
   { super(aCity, aStreet, anAvenue, aDirection);
     this.setIcon(new ArmRobotIcon());
   }
}
```

2.8 Patterns

In this chapter we've seen patterns to extend a class, write a constructor, and implement a parameterless command. These are all extremely common patterns; so common, in fact, that many experienced programmers wouldn't even recognize them as patterns. We've also seen a much less common pattern to draw a picture.

2.8.1 The Extended Class Pattern

Name: Extended Class

Context: You need a new kind of object to provide services for a program you are writing. An existing class provides objects with closely related services.

Solution: Extend the existing class to provide the new or different services required. For example, the following listing illustrates a new kind of robot that provides a service to turn around.

```
import becker.robots.*;
public class TurnAroundBot extends Robot
{
        public TurnAroundBot (City aCity, int aStreet,
                              int anAvenue, Direction aDirection)
        { super(aCity, aStreet, anAvenue, aDirection);
        }

        public void turnAround()
        { this.turnLeft();
          this.turnLeft();
        }
    }
```

This listing also makes use of the Constructor and Method patterns. More generally, a Java class uses the following code template:

```
    import «importedPackage»;     // may have 0 or more import statements

    public class «className» extends «superClass»
    { «list of attributes used by this class»
      «list of constructors for this class»
      «list of services provided by this class»
    }
```

The Java Program pattern can be seen as a special version of the Class pattern, which has no constructors or attributes and contains only the specialized service named `main`.

Consequences: Objects instantiated from a subclass respond to the same messages as objects instantiated from the superclass. Instances of the subclass may behave differently from instances of the superclass, depending on whether methods have been overridden. The subclass's objects may also respond to messages not defined by the superclass.

It should make sense for the client using the subclass to also use any of the methods in its superclass. If not, think carefully about the superclass; it may have been chosen incorrectly. If there is no class to serve as the superclass, use `Object`, a class that contains the minimal set of methods required of every Java class.

Related Patterns:

- The Constructor pattern is always applied within an instance of the Extended Class pattern.
- The Parameterless Command pattern is always applied within an instance of the Extended Class pattern.

2.8.2 The Constructor Pattern

Name: Constructor

Context: Instances of a class must be initialized when they are constructed.

Solution: Add a constructor to the class. A constructor has the same name as the class and is usually preceded by the keyword `public`. It often has parameters so that the client constructing the object can provide initialization details at run time. The constructor must also ensure that the object's superclass is appropriately initialized, using the keyword `super`. The types of the parameters passed to `super` should match the types required by one of the constructors in the superclass. Constructors and their parameters should always have a documentation comment.

Following is an example of a constructor that simply initializes its superclass with values received via its parameters:

```
/** Construct a new special edition robot.
*   @param aCity        The city containing the robot.
*   @param str          The robot's initial street.
*   @param ave          The robot's initial avenue.
*   @param dir          The robot's initial direction. */
public RobotSE(City aCity, int str, int ave, Direction dir)
{ super(aCity, str, ave, dir);
}
```

More generally, a constructor makes a call to super and may then execute other Java statements to initialize itself.

```
/**«Description of what this constructor does.»
* @param «parameterName» «Description of parameter»
*/
public «className»(«parameter list»)
{ super(«arguments»);
  «list of Java statements»
}
```

If the parameter list is empty, the documentation comment does not contain any `@param` tags. Otherwise, the documentation comment contains one `@param` tag for each parameter.

Consequences: A constructor should ensure that each object it creates is completely and consistently initialized to appropriate values. Doing so leads to higher quality software.

In some circumstances the compiler supplies a missing constructor, but don't rely on the compiler to do so. If you always supply a constructor, you increase your chances of remembering to initialize everything correctly. You also minimize the possibility that future changes will break your software.

Related Patterns: The Constructor pattern always occurs within a pattern for a class. The Extended Class pattern is one such pattern.

2.8.3 The Parameterless Command Pattern

Name: Parameterless Command

Context: You are writing or extending a class and need to provide a new service to clients. The service does not require any information other than the object to act upon (the implicit parameter) and does not return any information.

Solution: Use a parameterless command with the following form:

```
/** «Description of the command.»
*/
public void «commandName»()
{ «list of statements»
}
```

One example that implements this pattern is the `turnAround` method:

```
/** Turn the robot around to face the opposite direction. */
public void turnAround()
{ this.turnLeft();
  this.turnLeft();
}
```

Consequences: The new service is available to any client of objects instantiated from this class. In future chapters we will see replacements for the `public` keyword that make the command's use more restricted.

Related Patterns: The Parameterless Command pattern always occurs within a pattern for a class. The Extended Class pattern is one such pattern.

2.8.4 The Draw a Picture Pattern

Name: Draw a Picture

Context: You want to show an image to the user that is constructed from ovals, rectangles, lines, and strings.

Solution: Extend `JComponent` and override the `paintComponent` method to draw the image. Display the new component inside a frame using the Display a Frame pattern.

In general, the code for the extension of `JComponent` will be as follows:

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

public class «className» extends JComponent
{
        public «className»()
        { super ();
          this.setPreferredSize(
               new Dimension(«prefWidth», «prefHeight»));
        }


        // Draw the image.
        public void paintComponent(Graphics g)
        { super.paintComponent(g);

          «statements using g to draw the image»
        }
     }
```

Consequences: The component will display the image drawn in `paintComponent`, but it is not very smart about the image size and may give some strange results if the frame is resized. These issues will be addressed in Section 4.7.1.

Related Patterns:

- The extended component can be displayed using the Display a Frame pattern.
- The Draw a Picture pattern is a specialization of the Extended Class pattern and contains an example of the Constructor pattern.

2.9 Summary and Concept Map

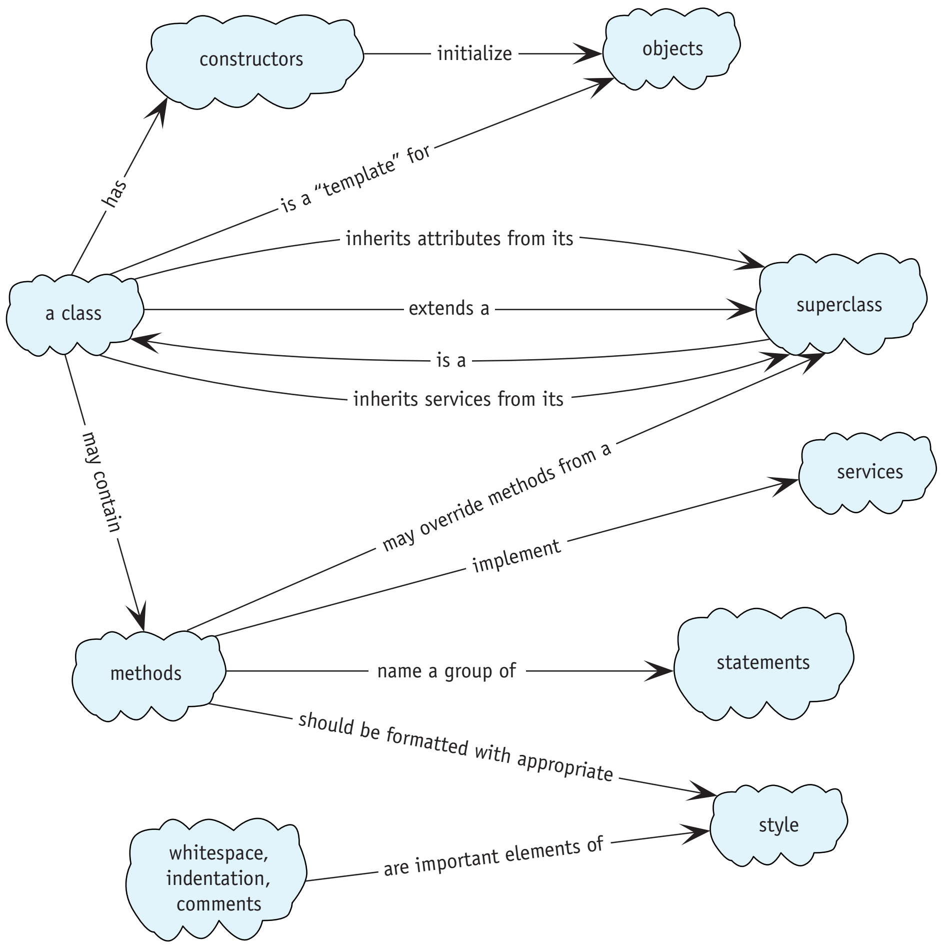
Extending an existing class is one way to customize a class with new or modified services. Instances of the resulting class (subclass) can be visualized as having an instance of the existing class (superclass) inside it. The subclass's constructor must ensure that the superclass is initialized by calling `super` with appropriate arguments.

The subclass inherits all of the methods from the superclass. New methods added to the subclass may call methods in the superclass or other methods in the subclass. When a message is sent to an object, the process of determining which method to execute is called method resolution. Method resolution always starts with the class used to instantiate the object, unless the method is called using `super` in which case method resolution begins with the superclass of the class making the call.

A method in the subclass overrides an existing method when it has the same name and parameter types as a method in one of the superclasses. The overridden method may be called using the keyword `super`.

The components used to display graphical user interfaces make extensive use of inheritance and overriding. For example, one overrides the `paintComponent` method to alter the appearance of a component.

Style is an important part of writing understandable programs. White space, indentation, and choice of identifiers all make a significant contribution to the overall clarity of the program.


constructors
initialize
objects
has
is a "template" for
inherits attributes from its
a class
extends a
superclass
is a
inherits services from its
may contain
may override methods from a
services
implement
methods
name a group of
statements
should be formatted with appropriate
style
whitespace, indentation, comments
are important elements of

2.10 Problem Set

Written Exercises

2.1 Based on what you now know about the `getSpeed` and `setSpeed` services from Section 2.6.1, revise the `Robot` class diagram shown in Figure 1-8.

2.2 Consider a robot that implements `turnRight` as shown in Listing 2-4 and implements `turnAround` by calling `turnRight` twice.

a. Describe what a robot executing this version of `turnAround` does.

b. How much time does this version of `turnAround` require compared to the version in Listing 2-4?

2.3 Write a new constructor for the `RobotSE` class. Robots constructed with this new constructor will always be placed at the origin of the city facing `EAST`.

2.4 Add arrows to Figure 2-19, which is similar to Figure 2-6, showing the following method calls:

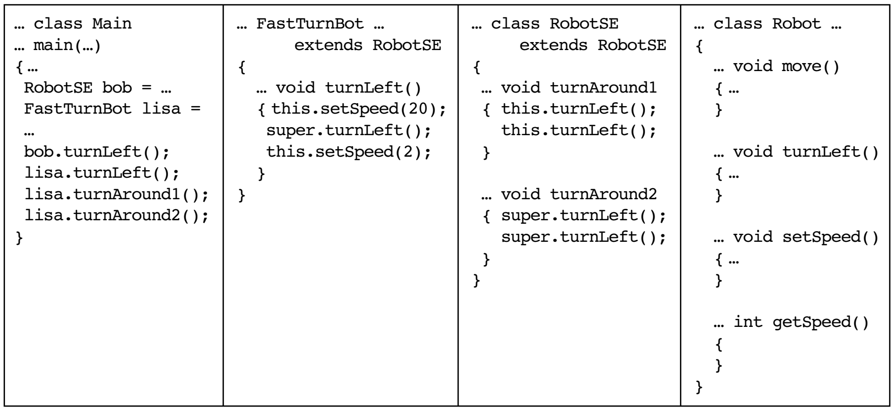
a. Method calls resulting from `bob.turnLeft()`

b. Method calls resulting from `lisa.turnLeft()`

c. Method calls resulting from `lisa.turnAround1()`

d. Method calls resulting from `lisa.turnAround2()`

To keep the diagrams uncluttered, answer each part of the question on a separate copy of the diagram and omit arrows the second time a method is called from the same place (for example, do not draw arrows for the second call to `turnLeft` in `turnAround1`).



```
… class Main
… main(…)
{…
 RobotSE bob = …
 FastTurnBot lisa =
 …
 bob.turnLeft();
 lisa.turnLeft();
 lisa.turnAround1();
 lisa.turnAround2();
}
```

```
… FastTurnBot …
      extends RobotSE
{
  … void turnLeft()
  { this.setSpeed(20);
   super.turnLeft();
   this.setSpeed(2);
  }
}
```

```
… class RobotSE
      extends RobotSE
{
 … void turnAround1
 { this.turnLeft();
   this.turnLeft();
 }

 … void turnAround2
 { super.turnLeft();
   super.turnLeft();
 }
}
```

```
… class Robot …
{
  … void move()
  { …
  }

  … void turnLeft()
  { …
  }

  … void setSpeed()
  { …
  }

  … int getSpeed()
  {
  }
}
```

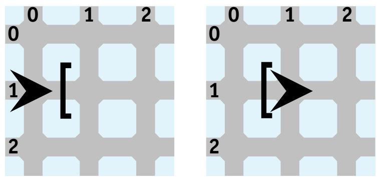
(figure 2-19)

Illustrating method calls and method resolution

2.5 The change from the initial situation to the final situation shown in Figure 2-20 is accomplished by sending the robot exactly one `move` message. Ordinarily such a stunt would cause the robot to crash. There are at least three fundamentally different approaches to solving this seemingly impossible problem. Explain two of them.

(figure 2-20)

A seemingly impossible situation



Initial situation Final situation

Programming Exercises

2.6 Write a new class, `MileMover`, that includes two methods: `moveMile` moves a robot forward 10 intersections, and `move1000Miles` which moves the robot forward 1,000 miles. Your solution should be *much* shorter than 1,000 lines of code.

2.7 Instances of the `BackupBot` class can respond to the `backup` message by moving to the intersection immediately behind it, facing their original direction.

a. Create the `BackupBot` class by extending `Robot`.

b. Arrange for the `backup` method to take the same amount of time as the `move` method.

c. Create the `BackupBot` class by extending `RobotSE` and taking advantage of the methods it contains.

2.8 Extend `RobotSE` to create a `LeftDancer` class. A `LeftDancer`, when sent a `move` message, ends up at the same place and facing the same direction as a normal robot. But it gets there more "gracefully." A `LeftDancer` first moves to the left, then forward, and then to the right.

2.9 Extend `RobotSE` to create a `TrailBot` class. A `TrailBot` leaves behind a trail of "crumbs" (`Thing` objects) whenever it moves. Arrange for instances of `TrailBot` to always start with 100 `Things` in its backpack. (*Hint*: Check out the `Robot` constructors in the documentation.)

a. Add a `trailMove` method. When called, it leaves a "crumb" on the current intersection and moves forward to the next intersection. `move` still behaves as usual.

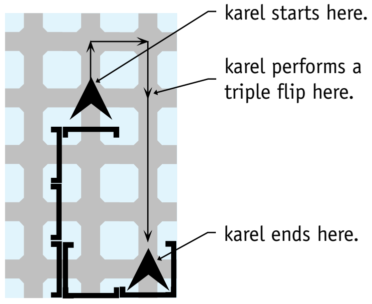
b. Arrange for a `TrailBot` to always leave a "crumb" behind when it moves.

2.10 Extend `Robot` to make a new class, `PokeyBot`. `PokeyBots` ordinarily make one move or turn every two seconds (that is, 0.5 moves per second). However, the statement `pokey.setSpeed(3)` makes a robot named `pokey` go faster until a subsequent `setSpeed` command is given. Write a program that instantiates a `PokeyBot` and verifies that it moves more slowly than a standard `Robot`. Also verify that `setSpeed` works as described. *Hint*: You do not need to override `move` or `turnLeft`.

2.11 Implement `turnLeft` as discussed in Section 2.6.1 but using `this.turnLeft()` instead of `super.turnLeft()`. Run a test program and describe what happens, using a diagram similar to Figure 2-5 to illustrate what happened. (*Hint*: You may not be able to read the first line of the resulting error message. It probably says something about "Stack Overflow," which means the computer ran out of memory. A little bit of memory is used each time a method is called until that method is finished executing.)

Programming Projects

2.12 `karel` the robot has taken up diving. Write a program that sets up the following situation with `karel` at the top of the diving board. The single message `dive` should cause it to do a triple front flip (turn completely around three times) at the location shown while it dives into the pool (see Figure 2-21). `karel` is an instance of the `Diver` class.



(figure 2-21)

Diving into a pool

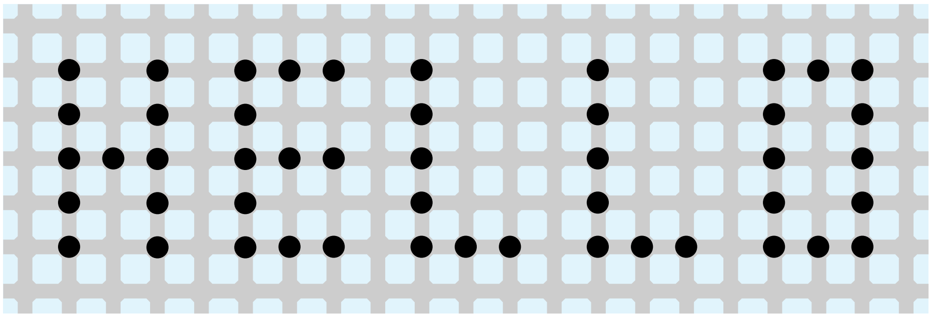
2.13 You would like to send a greeting to astronauts in the International Space Station orbiting Earth. Create a `WriterBot` class that can be used to "write" the message "Hello" by lighting bonfires (`Things`). The final situation is shown in Figure 2-22. The `WriterBot` class will contain a method to write each letter.

a. Write a `main` method that uses a single `WriterBot` to write the entire message. Instantiate the robot with at least 48 `Things` in its backpack. (Check the documentation for the `Robot` or `RobotSE` class for an alternate constructor.)

b. Write a `main` method that uses five `WriterBots`, one for each letter. Instantiate each robot with enough `Things` to write its assigned letter.

(figure 2-22)

Friendly message



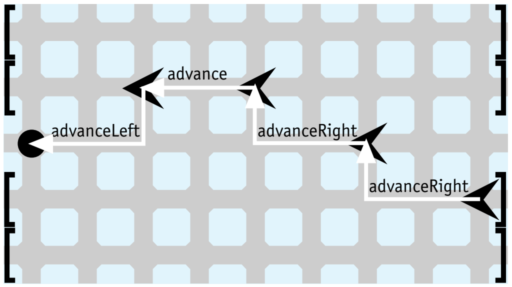
2.14 Write a program where robots practice playing soccer, as shown in Figure 2-23. Your program will have four `SoccerBot` robots. Each has methods to `advance`, `advanceLeft`, and `advanceRight`. Each of these methods begins with picking up the "ball" (a `Thing`), moving it in the pattern shown, and then dropping it.

a. Write a `main` method that sets up the city as shown and directs the four players to move the ball along the path shown by the arrows.

b. Write a subclass of `City`, `SoccerField`, that arranges the "goals" as shown. Add the ball and soccer players in the `main` method, as usual. (*Hint*: The `Wall` constructor requires a `City`. Which city? "this" city.)

(figure 2-23)

Soccer field with practicing robots



2.15 When a `Lamp` is on all that is visible is a soft yellow circle representing the "light." The "lamp" itself doesn't show unless it is off. Read the documentation for `CompositeIcon`. Then modify Listing 2-6 to make `Lamp` objects show both the "lamp" and the "light" when they are on.

2.16 In Section 2.3.2, we extended the `Thing` class, and in Section 2.2.4, we saw that `this` can be used to invoke methods inherited from the superclass.

a. Use these techniques to extend `JFrame` to obtain `CloseableJFrame`, which sets its default close operation, automatically opens to a default position and size of your choice, and is visible. Write a test program to ensure your new class works.

b. Modify your solution so the client creating the frame can specify its position and size via parameters.

2.17 Extend the functionality of `JFrame` so that a simple program containing the following code:

```
public static void main(String[] args)
{ ColorChooserFrame ccf = new ColorChooserFrame();
}
```

will cause a frame containing a `JColorChooser` to appear on the screen, appropriately sized. See Programming Project 1.18 for background.

2.18 Sketch a scene on graph paper that uses a combination of several rectangles, ovals, lines, and perhaps strings (text). Write a program that displays your scene.

2.19 Extend the `Icon` class as shown in Figure 2-24. The grid is to aid you in painting; it is not intended to be displayed by your icons.

a. Choose one of the icons shown in Figure 2-24 to implement. Instantiate a `Thing` and use its `setIcon` method to display your icon.

b. Introduce a `Car` class to the robot world. A `Car` is really a `Robot`, but uses a `CarIcon`, as shown in Figure 2-24.

c. Write a `TreeIcon` class that appears approximately as shown in Figure 2-24, using at least three shades of green. Extend `Thing` to make a `Tree` class and construct a city with several trees in it. Robots should not be able to pick up and move trees.

d. Create a `Lake` class that extends `Thing` and is displayed with a `LakeIcon` as shown in Figure 2-24. Robots should not be able to pick up and move lakes, and if they try to enter a lake, they break. Research the `Thing` class to discover how to implement these features.

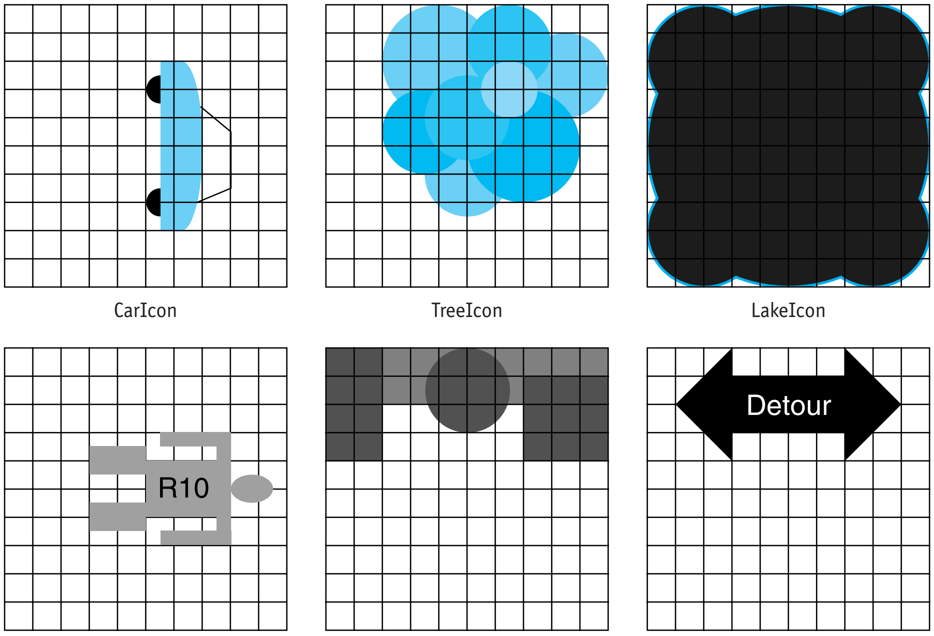
e. Write an `R10Icon` for an advanced type of `Robot`, as shown in Figure 2-24. Research the `setFont` method in the `Graphics` class and the `Font` class to label the robot. Construct a city with at least one robot that uses your icon.

f. Extend `Wall` to create a `Building` class. Use a `BuildingIcon`, as shown in Figure 2-24, to display your buildings. Robots should not be able to pass through or over your buildings.

g. Write a `DetourIcon`. Use it in `DetourSign`, an extension of `Thing`. You will need to research the `Polygon` class and then use the `fillPolygon` method in the `Graphics` class. Research the `Thing` class to learn how to make your sign block robots from entering or exiting the intersection on the `NORTH` side. You may assume that the sign will always be placed on the north side of the intersection.

(figure 2-24)

Patterns for icons


R10
Detour
CarIcon
TreeIcon
LakeIcon
R10Icon
BuildingIcon
DetourIcon