Chapter 1 Programming with Objects

Chapter Objectives

After studying this chapter, you should be able to:

- Describe models
- Describe the relationship between objects and classes
- Understand the syntax and semantics of a simple Java program
- Write object-oriented programs that simulate robots
- Understand and fix errors that can occur when constructing a program
- Read documentation for classes
- Apply the concepts learned with robots to display a window as used in a graphical user interface

A computer program usually models something. It might be the ticket sales for a concert, the flow of money in a corporation, or a game set in an imaginary world. Whatever that something is, a computer program abstracts the relevant features into a model, and then uses the model to help make decisions, predict the future, answer questions, or build a picture of an imaginary world.

In this chapter, we create programs that model a world filled with robots, directing them to move, turn, pick up, transport, and put down things. This robot world is simple to model, but quickly reveals key concepts of object-oriented programming: objects, classes, attributes, and services.

1.1 Modeling with Objects

Fifteen years ago I went to the local concert hall and asked the ticket agent for two tickets for the March 21 concert. "Where do you want to sit?" he asked.

"That depends on what's available," I answered.

The agent grabbed a printed map of the concert hall. It was clearly dated "March 21," noted the name of the performer, and showed a map of the auditorium's seats. Seats that had already been sold were marked with a red X. The seats were also color-coded: the most expensive seats were green, the moderately priced seats were black, and the least expensive seats were blue.

Fifteen years ago, the ticket agent showed me the map and stabbed his finger on a pair of seats. "These are the best seats left, but the choice is yours."

I quickly scanned the map and noticed that a pair of less expensive seats with almost the same sightlines was not far away. I chose the cheaper seats, and the agent promptly marked them with a red X.

Fast-forward fifteen years. Today I order tickets from the comfort of my home over the Web. I visit the concert hall's Web site and find the performance I want. I click the "purchase tickets online" button and am shown a color-coded map of the theatre. I click on the seats I want, enter my credit card information, and am assured that the tickets will be mailed to me promptly.

1.1.1 Using Models

A **model** is a simplified description of something. It helps us, for example, make decisions, predict future events, maintain up-to-date information, simulate a process, and so forth. Originally, the local concert hall modeled ticket sales with a simple paper map of the auditorium. Later, a Web-based computerized model performed the same functions—and probably many more.

KEY IDEA

A model is a simplified description of something.

To be useful, a model must be able to answer one or more questions. The paper-based model of ticket sales could be used to answer questions such as:

- What is the date of the concert?
- Who is playing?
- How many tickets have been sold to date?
- How many tickets are still unsold?
- Is the ticket for seat 22H still available?
- What is the price of the ticket for seat 22H?

- Which row has unsold tickets for 10 consecutive seats and is closest to the stage?
- What is the total value of all the tickets sold to date?

Models often change over time. For instance, the ticket sales model was updated with two new red X's when I bought my tickets. Without being updated, the model quickly diverges from the thing it represents and loses its value because the answers it provides are wrong.

We often speak of models or elements of a model as if they were real. When the ticket agent pointed to the map and said, "These are the best seats left," we both knew that what he was pointing at were not seats, but only images that *represented* actual seats. The model provided a correspondence. Anyone could use that model to find those two seats in the concert hall.

We often build models without even being aware of it. For example, you might make a mental list of the errands you want to run before having supper ready for your roommate at 6 o'clock, as shown in Figure 1-1: stopping at the library to pick up a book (10 minutes), checking e-mail on a public terminal at the library (5 minutes), and buying a few groceries (10 minutes). Checking your watch (it's 4:15) and factoring in 45 minutes for the bike ride home and 30 minutes to prepare supper, you estimate that you can do it all, with a little time to spare. It takes longer than expected, however, to find the book, and there's a line at the library checkout counter. The library errand took 20 minutes instead of 10. Now it's 4:35, and you must make some choices based on your updated model: have supper a little late, skip the e-mail, hope that you can cook supper in 25 minutes instead of 30, and so forth. You have been modeling your time usage for the next two hours.

(figure 1-1)

Sample schedule

```
4:15  Pick up library book.
4:25  Check e-mail.
4:30  Buy groceries.
4:40  Bike home.
5:25  Cook supper.
6:00  Supper.
```

KEY IDEA

Models focus on relevant features.

Models form an **abstraction**. Abstractions focus only on the relevant information and organize the remaining details into useful higher-level "chunks" of information. People can only manage about seven pieces of information at a time, so we must carefully choose the information we manage. By using abstraction to eliminate or hide some details and group similar details together into a chunk, we can manage more complex ideas. Abstraction is the key to dealing with complexity.

For example, the ticket sales model gives ticket buyers and agents information about which tickets are available, where the corresponding seats are located, and their price. These were all relevant to my decision of which tickets to purchase. The map did not

provide information about the seat's fabric color, and I really didn't care, because that was irrelevant to my decision. Furthermore, the color-coding of the concert hall map conveniently chunked information, which helped me make a decision quickly. It was easier to see all the least expensive seats in blue rather than consulting a long list of seat numbers.

Beyond information, models also provide operations that can be performed on them. In the concert hall model operations include "sell a ticket" and "add a new concert." In the informal time management model for errands, operations include "insert a new errand," "drop an errand from the list," and "recompute the estimated start time for each errand."

1.1.2 Using Software Objects to Create Models

The concert hall's computer program and the paper map it replaced model the ticket-selling problem using different technologies. One uses pre-printed sheets of paper marked with a simple X. The other involves a computer with a detailed set of instructions, called a **program**. If the program is written in an **object-oriented programming language**, such as Java, the computer program uses cooperating **software objects**. A software object usually corresponds to an identifiable entity in the problem. The concert hall program probably has an object modeling the concert hall's physical layout, a collection of objects that each models a seat in the concert hall, and another collection of objects that each models an upcoming concert. A program maintaining student enrollments in courses would likely model each student with an object, and use other objects to model each course.

KEY IDEA

Object-oriented programs use software objects to model the problem at hand.

KEY IDEA

Computer science has a specialized vocabulary to allow precise communication. You must learn this vocabulary.

Each of the software objects can perform tasks such as:

- Maintain information about part of the problem the program models.
- Answer questions about that part of the problem based on the information it maintains.
- Change its information to reflect changes in the real-world entity it models.

The information kept by the object is called its **attributes**. Objects respond to **queries** for information and to **commands** to change their attributes. Queries and commands are collectively referred to as **services**. An object provides these services to other objects, called **clients**. The object providing the service is called, appropriately, the **server**. We will explore these concepts in the coming pages.

KEY IDEA

Server objects provide services—queries and commands—to client objects.

Queries and Attributes

Queries are the questions to which an object can respond. A query is always answered by the object to which it is directed. It might be true or false ("Is the ticket for seat 22H still for sale?"), a number ("How many tickets have been sold?"), a string of characters

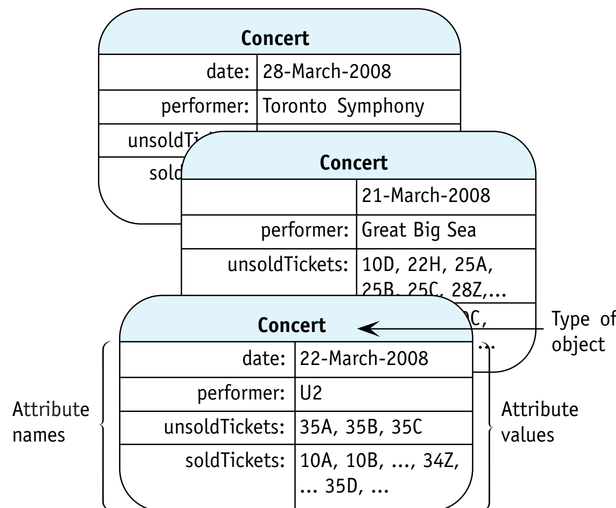
("What is the name of the band that is playing?"), or even another object such as a date object ("What is the date of the concert?"). Queries are said to `return` answers to their clients. An object can't respond to just any query, only to those it was designed and programmed to support.

KEY IDEA

Objects have attributes. Answers to queries are based on the values of the attributes.

The answers provided by queries are always based on the object's attributes. If an object must answer the query, "What is the date of the concert?" then it must have an attribute with information about the date. Similarly, if it must answer the question, "How many tickets have been sold to date?" it must have an attribute that has that information directly, or it must have a way to calculate that information, perhaps by counting the number of tickets that have been sold. Information about which tickets have been sold would be kept in an attribute.

The concert hall's program must model ticket sales for many concerts, each represented by its own concert object. If we look at several concert objects, we'll notice they all have the same set of attributes, although the values of those attributes may be different. One way to show the differing attribute values is with an **object diagram**, as shown in Figure 1-2. Each rounded rectangle represents a different concert object. The type of object is shown at the top. Below that is a table with attribute names on the left and attribute values on the right. For example, the attribute "date" has a value of "21-March-2008" for one concert. That same concert has the value "Great Big Sea" for the "performer" attribute.



(figure 1-2)

Object diagram showing three concert objects with their attributes

One analogy for objects is that an object is like a form, such as an income tax form. The government prints millions of copies of the form asking for a person's name, address, taxpayer identification number, earned income, and so forth. Each piece of information is provided in a little box, appropriately labeled on the form. Each copy of the form starts like all

the others. When filled out, however, each form has unique values in those boxes. It could be that two people have exactly the same income and birthday, with the result that some forms have the same values in the same boxes—but that's only a coincidence.

Just as each copy of that tax form asks for the same information, every concert object has the same set of attributes. Each copy of the form is filled out with information for a specific taxpayer; likewise, each concert object's attributes have values for a specific concert. In general, there may be many objects of a given type. All have the same set of attributes, but probably have different values for those attributes.

KEY IDEA

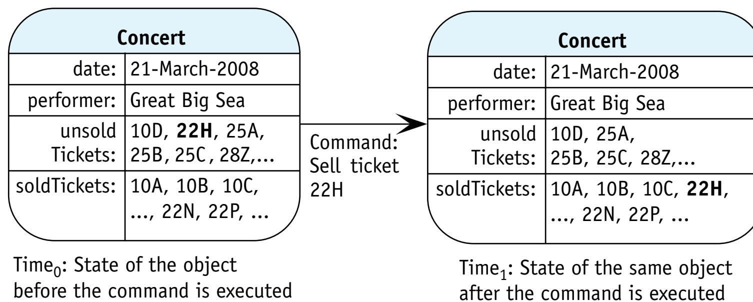
Every object of a given type has the same set of attributes, but usually has different values for the attributes.

Commands

When a ticket is sold for seat 22H for the March 21 concert, the appropriate concert object must record that fact. This record keeping is done with a command. The object is "commanded" to change its attributes to reflect the new reality. This change can be visualized with a **state change diagram**, as shown in Figure 1-3. A state change diagram shows the state of the object before the command and the state of the object after the command. The **state** is the set of attributes and their values at a given point in time. As time passes, it is normal for the state of an object to change.

KEY IDEA

Commands change the state of the object.



(figure 1-3)

State change diagram showing the change in state after a command to sell seat 22H is given to a concert object

Classes

When we write a Java program, we don't write objects, we write classes. A **class** is a definition for a group of objects that have the same attributes and services. A programmer writing the concert hall program would write a concert class to specify that all concert objects have attributes storing the concert's date, performers, and so on. The class also specifies services that all concert objects have, such as "sell a ticket," and "how many tickets have been sold?"

KEY IDEA

A Java programmer writes a class by specifying the attributes and services the classes' objects will possess.

Once a concert class is defined, the programmer can use it to create as many concert objects as she needs. Each object is an **instance**, or one particular example, of a class. When an object is first brought into existence, we sometimes say it has been **instantiated**.

The distinction between class and object is important. It's the same as the distinction between a factory and the cars made in the factory, or the distinction between a cookie cutter and the cookies it shapes. The pattern used to sew a dress is different from the dress produced from it, just as a blueprint is different from the house it specifies. In each case, one thing (the class, factory, or cookie cutter) specifies what something else (objects, cars, or cookies) will be like. Furthermore, classes, factories, and cookie cutters can all be used to make many instances of the things they specify. One factory makes many cars; one class can make many objects. Finally, just as most of us are not interested in cookie cutters for their own sakes, but in the cookies made from them, our primary interest in classes is to get what we really want: software objects that help model some problem for us.

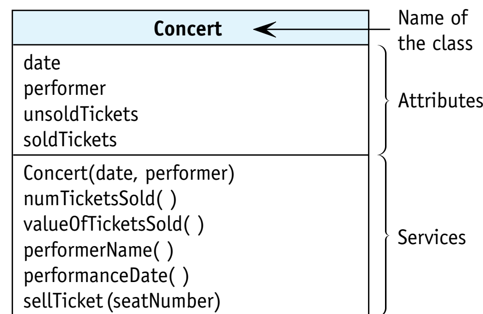
Class Diagrams

KEY IDEA

A class diagram summarizes all of the objects belonging to that class.

Just as architects and dress designers communicate parts of their designs visually through blueprints and patterns, software professionals use diagrams to design, document, and communicate their programs. We've already seen an object diagram in Figure 1-2 and a state change diagram (consisting of two object diagrams) in Figure 1-3. Another kind of diagram is the **class diagram**. Class diagrams show the attributes and services common to all objects belonging to the class. The class diagram for the concert class summarizes all the possible concert objects by showing the attributes and services each object has in common with all other concert objects.

A class diagram is a rectangle divided into three areas (see Figure 1-4). The top area contains the name of the class. Attributes are named in the middle area, and services are in the bottom area.



(figure 1-4)

Class diagram for the Concert class showing four attributes and six services

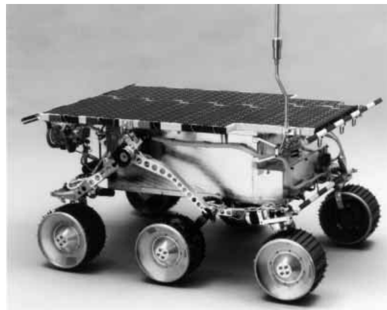
1.1.3 Modeling Robots

Every computer program has a model of a problem. Sometimes the problem is tangible, such as tracking concert ticket sales or the time required to run errands before supper. At other times, the problem may be more abstract: the future earnings of a

company under a given set of assumptions or the energy loss of a house. Sometimes the problem is to visualize a figment of someone's imagination, such as a game set on a far-off world at some point in the future.

Many of the programs in this textbook model imaginary robots and the city in which they operate. The programs cause robots to move on the computer screen as they perform various tasks. This model was chosen to be basic enough to grasp easily, yet complex enough to be interesting; simple enough to be easy to program, yet rich enough to show many important object-oriented concepts. The robots and their world are described in Section 1.2. Section 1.3 describes using software objects to model the robots, and Section 1.4 will present the first program.

The robots our programs model are similar to the small robotic explorers NASA landed on Mars. The first, named Sojourner, landed on Mars on July 4, 1997. It could move around the Martian landscape, take photographs, and conduct scientific experiments. Sojourner was about two feet long and could travel at a top speed of two feet per minute. A photo of the explorer is shown in Figure 1-5.



(figure 1-5)

Sojourner, a robotic explorer landed on Mars by NASA

Sojourner was controlled from Earth via radio signals. Because radio signals take approximately 11 minutes to travel from Earth to Mars, Sojourner could not be controlled in real time. (Imagine trying to drive a car with a minimum of 22 minutes elapsing between turning the steering wheel and receiving feedback about the change in direction.) Instead, controllers on Earth carefully mapped out the movements and tasks Sojourner was to do, encoding them as a sequence of messages. These messages were sent to Sojourner, which then attempted to carry them out. Feedback regarding the entire sequence of messages was sent back to Earth, where controllers then worked out the next sequence of messages.

Sojourner had a computer on board to interpret the messages it received from Earth into electrical signals to control its motion and scientific instruments. The computer's processor was an Intel 80C85 processor containing only 6,500 transistors and executing about 100,000 instructions per second. This processor was used almost 15 years earlier in the Radio Shack TRS-80 home computer.

In contrast, a top-of-the-line Pentium processor in 1997 had about 7.5 million transistors and executed about 300,000,000 instructions per second.

Why did Sojourner use such a primitive processor? The 80C85 consumes tiny amounts of power compared with its state-of-the-art cousins and is much more likely to operate correctly in the presence of cosmic rays and extreme temperatures.

1.2 Understanding Karel's World

KEY IDEA

Karel's world is one of the "realities" we will be modeling with software.

The city where `karel`[1] the robot exists is pretty plain. It includes other robots, with a range of capabilities. It also includes intersections connected by avenues and streets on which robots travel, and where there may be several kinds of things. However, the city does not include office buildings, restaurants, traffic lights, newspaper dispensers, or homes. As you learn to program, you may want to change that fact.

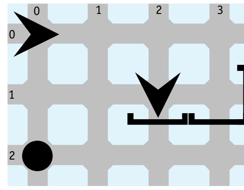
1.2.1 Avenues, Streets, and Intersections

What a city does have are **roads**. Some roads, called **streets**, run east and west, while other roads, called **avenues**, run north and south. (A helpful way to remember which is which is that the "A" and "v" in "Avenue" point up and down—or north and south on a map—whereas the cross strokes of the "t"s in "Street" run east and west.)

Streets and avenues are both numbered starting with 0. This convention is unusual among urban planners, but normal among Java programmers. Street 0 is located on the north (top) side, while Avenue 0 runs along the west (left) side. The place where these two roads meet is called the **origin**.

Figure 1-6 shows a small portion of a city with a robot facing east at the origin and another facing south at the intersection of 1st Street and 2nd Avenue. We can use a shorthand notation for specifying intersections. Instead of "1st Street and 2nd Avenue," we can write (1, 2). The first number in the pair gives the street, and the second gives the avenue.

[1] We will often name robots "karel" (pronounced "kär-əl"—the same as "Karl" or "Carl") in recognition of the Czechoslovakian dramatist Karel Capek (1890–1938), who popularized the word *robot* in his 1921 play R.U.R. (Rossum's Universal Robots). The word *robot* is derived from the Czech word *robota*, meaning "forced labor." The name is lowercase, in keeping with Java style.



(figure 1-6)

Small city with two robots, one at the origin facing east and one at the intersection of 1st Street and 2nd Avenue facing south

Intersections are unusually wide. Many robots can be on the same intersection at the same time without interfering with each other.

1.2.2 Walls and (other) Things

Intersections may be surrounded by **walls** on one or more sides. A wall stands at an edge of an intersection and blocks robots from entering or leaving the intersection in that direction. Robots can't push walls out of the way. A small extension on the end of each wall extends toward the intersection containing the wall.

The city shown in Figure 1-6 contains three walls, including two at the edges of the intersection at (1, 3). Another wall is immediately in front of the robot at (1, 2) and blocks it from proceeding south. The robot may go around the wall, of course.

Intersections may also have nondescript things. They are purposefully nondescript so we can imagine them to be whatever we want them to be: newspapers, lights, pieces of carpet, or flags. One such thing appears at (2, 0) in Figure 1-6. Robots can usually pick a thing up, put it in a backpack to carry it somewhere else, and then put it down again.

Eventually we will learn how to define classes of things with different appearances and services. Two examples already exist: flashers, like you might find marking a construction site, and streetlights.

1.2.3 Robots

Robots exist to serve their clients. The four services they perform most often are moving, turning, picking things up, and putting things down. Some additional services robots provide include answering queries about their location and direction, and responding to a command controlling their speed.

These are primitive services. Clients using a robot must give many small instructions to tell the robot how to perform a task. Beginning with Chapter 2, we will learn how to create new kinds of robots that provide services tailored to solving the problem at hand.

KEY IDEA

Software objects, such as robots, do things only in response to messages.

Robots don't do anything of their own volition. They respond only to **messages** sent to them from outside themselves. A robot performs a service only when it is invoked by a corresponding message.

In the following sections, we will look at these services in more detail.

Turning

When a robot receives a `turnLeft` message, it responds by turning left 90 degrees. When a robot facing north receives the `turnLeft` message, it turns to face west. A south-facing robot responds to a `turnLeft` message by turning to face east. When a robot turns, it remains on the same intersection.

Robots always start out facing one of the four compass points: north, south, east, or west. Because robots can turn only in 90-degree increments, they always face one of those four directions (except while they are in the act of turning).

LOOKING AHEAD

In Chapter 2, we will create a new kind of robot that responds to a `turnRight` message.

Robots do not have a `turnRight` instruction because it is not needed; three `turnLeft` messages accomplish the same task.

Turning is a safe activity. Unlike moving, picking things up, or putting things down, nothing can go wrong when turning.

Moving

When a robot receives a `move` message, it attempts to move from its current intersection to the next intersection in the direction it is facing. It remains facing the same direction. Robots can't stop between intersections; they are either on an intersection or in the process of moving to another one.

Things can go wrong when a robot receives a `move` message. In particular, if there is a wall immediately in front of a robot, moving causes that robot to break. When a robot breaks, it is displayed in three pieces, as shown in Figure 1-7, an error message is printed on the screen, and the program halts. An example of the error message is shown in Figure 1-20.

(figure 1-7)

When a robot facing a wall receives a `move` command, it crashes into the wall, breaks, and can no longer respond to commands

Robot facing a wall

Receiving a `move` command and crashing

Handling Things

When a robot receives a `pickThing` message, it attempts to pick up a thing from its current intersection. If there are several things the robot could pick up, it randomly chooses one of them. Robots have a backpack where they carry the things they pick up. Things are small and the backpack is large, so many things fit in it. Robots can also put things down in response to the `putThing` message.

As you might expect, a robot can experience difficulties in handling things. If a robot receives a `pickThing` message when there is nothing to pick up on the current intersection, the robot breaks. Similarly, when a robot receives a `putThing` message and its backpack is empty, the robot breaks. As with moving, after such a malfunction the robot appears damaged, an error message is printed, and the program halts.

LOOKING AHEAD

In Chapter 4 we will learn how to write programs where robots can detect if something can be picked up.

1.3 Modeling Robots with Software Objects

Not surprisingly, the software we will use to model robots mirrors the description in the previous section in many ways. Software objects model intersections, robots, walls, and things.

The software does not actually control real, physical robots that you can touch. Instead, it displays images of robots on the computer screen. The programs we will write cause the images to move about the city (also displayed on the screen) and perform various tasks. These programs are only useful in that they provide an excellent way to learn how to program a computer. You can transfer the knowledge you gain in writing robot programs to writing programs that model the problems that concern you.

As shown earlier in Figure 1-4, we can summarize objects with a class diagram that shows the attributes and services of each object belonging to the class. A class diagram for the `Robot` class is shown in Figure 1-8. The class diagram shows the four services discussed earlier, along with a special service to construct `Robot` objects.

Robot
int street int avenue Direction direction ThingBag backpack
Robot(City aCity, int aStreet, int anAvenue Direction aDirection) void move( ) void turnLeft( ) void pickThing( ) void putThing( )

(figure 1-8)

Incomplete class diagram for the `Robot` class

1.3.1 Attributes

Recall that the middle section of the class diagram lists the attributes. From the `Robot` class diagram, we can infer that each `Robot` object has four attributes. We might guess that the two named `street` and `avenue` record the street and avenue the robot currently occupies, and that `direction` records the direction it is facing. Finally, the `backpack` attribute might plausibly be where each robot keeps track of the things it is carrying. We can't know any of these details with absolute certainty, but it makes sense given what we know about the robot world described in Section 1.2, "Understanding Karel's World," and from the names of the attributes themselves.

LOOKING AHEAD

Attributes can have types other than `int`. See Chapter 7.

Preceding the names of the attributes is the type of information to which they refer. The **type** specifies the set of valid values for the attribute. The `street` and `avenue` attributes are preceded by `int`, which is Java shorthand for "integer." This information makes sense because we have been referring to streets and avenues with integers, such as 0, 1, or 5, but never with real numbers, such as 3.14159.

LOOKING AHEAD

The type of an attribute can be the name of a class, like `ThingBag`. See Chapter 8.

The type of `backpack` is a `ThingBag`. `ThingBags` can store a variable number of `Thing` objects. This attribute illustrates that a robot object makes use of other objects—these objects cooperate to model the problem.

KEY IDEA

Class diagrams are designed to help you understand a class. They may omit low-level details in the interest of clarity.

Sometimes a class diagram does not include all of the attributes. Why? The important part of a class is the services its objects provide—the things they can do. It is appropriate to say that the programmer implementing the class needs to know the attributes, but it's no one else's business how the object works internally. Nevertheless, we will find it helpful in discussing the services to know what attributes they need to maintain. The class diagram shown earlier in Figure 1-8 occupies a middle ground. It shows attributes that contribute to understanding the class, but omits others that don't, even though they are necessary to implement the class.

1.3.2 Constructors

The `Robot` class diagram lists five services: `Robot`, `move`, `turnLeft`, `pickThing`, and `putThing`.

KEY IDEA

Constructors create new objects. Services are performed by an object that already exists.

The first, `Robot`, is actually a constructor rather than a service, but is listed here for convenience. Although constructors have some similarities to services, there are important differences. The key difference is their purposes: services are performed by an object for some client, while constructors are used by a client to construct a new object. (Recall that the client is the object using the services of the `Robot` object.) Constructors always have the same name as the class.

When a new object is constructed, its attributes must be set to the correct initial values. The initial position of the robot is determined by the client. The client communicates the desired

initial position to the constructor by providing **arguments**, or specific values, for each of the constructor's **parameters**. The parameters are shown in the class diagram between parentheses: `Robot(City aCity, int aStreet, int anAvenue, Direction aDirection)`. Notice that there is a remarkable similarity between the constructor's parameters and the classes' attributes.

KEY IDEA

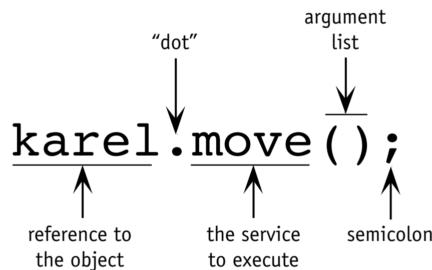
The constructor is responsible for correctly initializing the attributes.

1.3.3 Services

Suppose we have a robot that we refer to as `karel`. We can tell `karel` what to do by sending it **messages** such as `move` or `turnLeft`. A message requests that the object perform one of its services for a client, the sender of the message. The services in the `Robot` class diagram tell us which messages we can send to a robot object. A message to `karel` contains the name `karel`, a dot, and the name of a service followed by an argument list and a semicolon, as shown in Figure 1-9.

KEY IDEA

A client requests a service from an object by sending a message to the object.



(figure 1-9)

Details of sending a message to an object referred to as `karel`

In this message, `karel` identifies who is supposed to move. We don't want to move any robot (or vehicle or cow or anything else that can move), only the particular robot known as karel. Stating the object first is like having a conversation in a group of people. When you speak to a specific person within the group, you often start by saying his or her name—"Karel, please pass the potatoes." Because a program almost always contains many objects, identifying the recipient of the message is a requirement.

KEY IDEA

A message is always sent to a specific object.

After referring to the object, we place a dot, which connects the object to the message, `move`. The message must be one of the services the object knows how to perform—a service listed in the class diagram. Sending the message "jump" to `karel` would result in an error because `karel` does not have a jumping service.

KEY IDEA

Each message must correspond to one of its recipient's services.

Like the constructor, a service may have a list of parameters to convey information the object needs to carry out the service. Parameter lists always begin and end with parentheses. None of the four `Robot` services listed require additional information, and so all their parameter lists are empty (but the parentheses must still be present). Consequently, when the corresponding messages are sent to a `Robot` object no arguments are needed, although parentheses are still required.

Finally, the message ends with a semicolon.

When a robot receives the `move` message, it moves. It also updates the `street` and `avenue` attributes to reflect its new location. If `karel` is standing at 2nd Street and 1st Avenue facing south, `street` and `avenue` contain 2 and 1, respectively. As the `move` service is executed, `street` is updated to 3, but `avenue` remains 1. The `Robot` object also sends many messages to other objects in the program to make an image move on the computer's screen.

KEY IDEA

Commands are preceded by the word `void` in class diagrams.

The `move` service is preceded in the class diagram with the word `void`. This word means that `move` is a command that changes the state of a robot object rather than a query that answers a question. If it were a query, `void` would be replaced with the type of the answer it returns—an integer, a real number, or a string of characters, for example. Using the keyword `void` to mean "returns no answer" can be related to an English meaning of the word: "containing nothing."

Command Invocation

Invoking the remaining services listed in the class diagram (`turnLeft`, `pickThing`, and `putThing`) follows the same pattern as `move`. Start with a reference to a specific robot. Then add a dot, the message you want to send the robot, an empty argument list, and a semicolon. The designated robot responds by turning, picking, or putting, as described earlier. Furthermore, the services of any other class are invoked by following this same pattern. Not only the `Robot`, `Wall`, and `Thing` classes, but also classes modeling students or employees or printers or checkbooks or concerts follow this pattern. *All* objects follow this pattern.

Learning to recognize common patterns is an important part of becoming a good programmer. When this book uses a common pattern, a pattern icon appears in the margin, as shown beside the previous paragraph. A section near the end of each chapter explains the patterns in detail and generalizes them to be more broadly applicable. The first such section is Section 1.7.

1.4 Two Example Programs

It's time to put all this background to use. You know about the program's model, you know about classes and objects, and you know how to send an object a message to invoke one of its services. In this section, we'll take a look at a computer program that uses these concepts to accomplish a task.

1.4.1 Situations

When writing a program (or reading a program someone else has written), you must understand what the program is supposed to do. For our first program, let's imagine that a delivery robot is to pick up a parcel, represented by a `Thing`, at intersection (1, 2) and

deliver it to (2, 3). The **initial situation**, shown in Figure 1-10, represents the state of the city before the robot does its task. The **final situation**, also shown in Figure 1-10, is how we want the city to appear after the task is done.

KEY IDEA

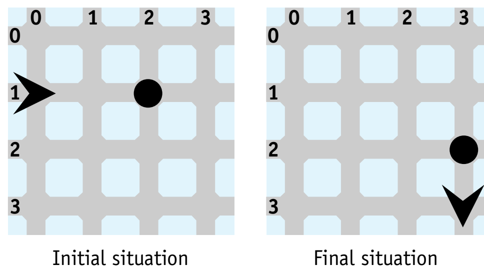
Many robot tasks can be specified by showing the way things are at the beginning and how we want things to end up.

This task could be accomplished in many ways. Perhaps the simplest is for the robot to perform the following steps:

move forward until it reaches the parcel
pick up the parcel
move one block farther
turn right
move a block
put the parcel down
move one more block

PATTERN

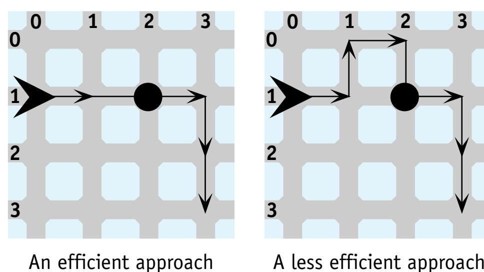
Sequential Execution



(figure 1-10)

Initial and final situations of a task to pick up and deliver a parcel

This path is illustrated on the left side of Figure 1-11. A more roundabout path is shown on the right. The roundabout path also accomplishes the task but results in a less efficient solution. If the robot were real, which solution would cause the robot to use the least power from its battery pack?



(figure 1-11)

Two approaches for the robot to perform the delivery task

Obviously, the robot could take any one of many possible paths to solve this problem. The following program takes the more efficient approach outlined in Figure 1-11.

1.4.2 Program Listing

LOOKING BACK

Before you can run the program in Listing 1-1, you need to have software installed on your computer. See the Preface for instructions.

Listing 1-1 shows the **source code** of a program to carry out the task just described. The source code contains the words and other symbols we write to instruct the computer.

Based on the previous discussion, you should be able to read the main body of the program and have a feel for how it works. Of course, you won't understand everything now, but it will all be explained in due course. You may be interested in knowing that much of the code in Listing 1-1 is repeated in every Java program and even more is repeated in every program using robots.

The line numbers on the left are not part of the program. They are included in the listing only so we can easily refer to specific parts of the program.

This program divides naturally into three parts: the code in lines 8–10, which constructs objects to set up the initial situation, the code in lines 13–22, which sends messages directing the robot to the final situation, and the remaining "housekeeping" required by the Java language. This division is reinforced by the comments written by the programmer at lines 7 and 12.

At a lower level of detail, Table 1-1 describes the purpose of each line of code. Use it to get a feel for the kinds of information present in a Java program, but don't expect to understand it all this early in the book. Lines 8–22 are the most important for right now; all will be discussed in detail later in the book.

The source code for the program in Listing 1-1 is available from the Robots Web site. Download the file `examples.zip`. After saving and expanding it, look in the directory `ch01/deliverParcel/`.

ch01/deliverParcel/

Listing 1-1: *A program to move a* `Thing` *from (1, 2) to (2, 3)*

```
1  import becker.robots.*;
2
3  public class DeliverParcel
4  {
5    public static void main(String[] args)
6    {
7      // Set up the initial situation
8      City prague    = new City();
9      Thing parcel   = new Thing(prague, 1, 2);
10     Robot karel    = new Robot(prague, 1, 0, Direction.EAST);
11
12     // Direct the robot to the final situation
13     karel.move();
```

Listing 1-1: *A program to move a* `Thing` *from (1, 2) to (2, 3)* (continued)

```
14      karel.move();
15      karel.pickThing();
16      karel.move();
17      karel.turnLeft();        // start turning right as three turns lefts
18      karel.turnLeft();
19      karel.turnLeft();        // finished turning right
20      karel.move();
21      karel.putThing();
22      karel.move();
23    }
24  }
```

(table 1-1)

An explanation of Listing 1-1

Line	Purpose
1	Makes code written by other programmers, such as the `Robot` class, easily available.
2, 11	Blank lines often add clarity for a person reading the program, but do not affect its execution in any way.
3	Identifies the class being written with the name `DeliverParcel`.
4, 6, 23, 24	Java uses braces to give structure to the program. The braces at lines 4 and 24 contain all the code belonging to the class. The braces at lines 6 and 23 contain all the code belonging to the service named `main`.
5	Identifies where the program will begin execution. Every program must have a line similar to this one.
7, 12	Text between two consecutive slashes and the end of the line is a **comment**. Comments are meant to help human readers and do not affect the execution of the program in any way.
8–10	Construct the objects required by the program.
13–22	Messages telling the robot named `karel` which services it should perform.

We now turn to a detailed discussion of lines 8–22. The remainder of the program will be discussed in Section 1.4.7.

1.4.3 Setting up the Initial Situation

The initial situation, as shown in Figure 1-10, is set up by constructing three objects in lines 8-10. The `City` object corresponds to all the intersections of streets and avenues. The `Robot` and `Thing` objects obviously correspond to the robot and thing shown in the initial situation.

Object Instantiation

LOOKING AHEAD

We will see similar declarations in other situations. All have the type first, then the name.

Each of the four statements has a similar structure. Consider line 10 as an example:

```
10     Robot karel   = new Robot(prague, 1, 0, Direction.EAST);
```

On the left side of the equal sign (=) is a **variable declaration**. A variable declaration first states the type of the object—in this case `Robot`—and then the name of the variable being declared, `karel`. A **variable** uses a name (`karel`) to refer to a value (in this case a `Robot` object), allowing the value to be used easily in many places in the program. The choice of variable name is up to the programmer. A meaningful name helps the understanding of people reading the program, including the programmer.

The object is instantiated on the right side of the equal sign. The keyword `new` signals that a new object will be constructed. After `new`, a constructor is named, in this case, `Robot`. It must be compatible with the type of the variable on the left side of the equal sign. For now, "compatible" means the two are identical. Eventually, we will ease this restriction.

When an object is constructed, the client object may need to provide information for the constructor to do its job. In this case, the client specifies that the new robot is to be created in the city named `prague` at the intersection of Street 1 and Avenue 0, facing east. Recall the values or arguments provided at line 10:

```
10     Robot karel   = new Robot(prague, 1, 0, Direction.EAST);
```

They correspond to the parameters shown in the class diagram:

```
Robot(City aCity, int aStreet, int anAvenue,
      Direction aDirection)
```

The arguments list the city first, then the street, avenue, and direction—in that order. Furthermore, the types of the values provided match the types given in the parameter list. `prague` refers to a `City` object, just as the parameter list specifies what the first value must be. Similarly, the second and third values are integers, just as the types for avenue and street specify.

LOOKING AHEAD

We will learn how to define our own sets of values in Chapter 7.

The type of the `Robot` constructor's last parameter is `Direction` and the value passed to it in line 10 of Listing 1-1 is `Direction.EAST`. `Direction` is a class used to define values with program-specific meanings. `EAST` is one of those special values. It should come as no surprise that `WEST`, `NORTH`, and `SOUTH` are other values defined by the `Direction` class. When one of these values is used in a program, its defining class, `Direction`, must accompany it.

Finally, line 10 ends with a semicolon (`;`), which marks the end of the statement. The function of semicolons in Java is similar to periods marking the end of English sentences.

1.4.4 Sending Messages

Lines 13–22 in Listing 1-1 direct the robot from the initial situation to the final situation. They give a precise sequence of instructions the robot must perform to accomplish the task. If the instructions were performed in a different sequence, the problem is unlikely to be solved correctly.

Each instruction follows the command invocation pattern observed earlier: give the name of the object being addressed, a dot, and then the service desired from that object. For example, `karel.move();` instructs the robot `karel` to execute its `move` service. The result is that the robot moves from one intersection to the next intersection in the direction it is currently facing (unless something blocks its way).

Command Invocation
Sequential Execution

Each of these instructions is one **statement** in the program. Lines 8-10 are another kind of statement, **declaration statements**, because they declare variables and assign initial values to them. In time, we will learn other kinds of statements that allow the program to make decisions, execute other statements repeatedly, and so on. Recall that all statements end with a semicolon.

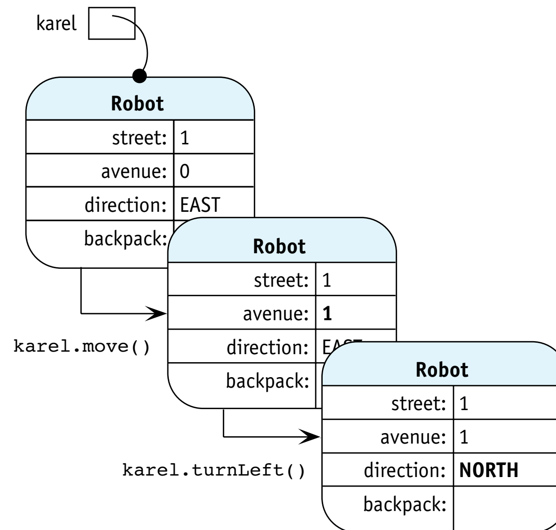
1.4.5 Tracing a Program

Simulating the execution of a program, also known as **tracing** a program, is one way to understand what it does and verify that the sequence of statements is correct. Tracing a program is like building a state change diagram, such as the one shown in Figure 1-12, for each statement in the program. Notice that the diagram covers two statements and shows the state before each statement and after each statement.

A program almost always involves many objects, so this can involve tracking a lot of information. Also, a formal state change diagram is difficult to draw and maintain. To overcome these problems, it's best to decide at the outset what information is relevant and to organize it in a table. It seems that the relevant information in the `DeliverParcel` program (Listing 1-1) includes the state of the robot and the state of the parcel, represented by a `Thing`. For the state of the robot, we will have four columns, one for each of the four attributes in the class diagram shown in Figure 1-8. For the parcel we will use a column for its street and another for its avenue. These are listed as headings in Table 1-2. If the program contained another robot, we would trace it by adding another group of four columns to the table.

(figure 1-12)

State change diagram tracing the execution of two statements



(table 1-2)

A table recording the change of program state while tracing the execution of the `DeliverParcel` *program in Listing 1-1*

Program Statement	karel street	karel avenue	karel direction	karel backpack	parcel street	parcel avenue
	1	0	east		1	2
13 `karel.move();`						
	1	1	east		1	2
14 `karel.move();`						
	1	2	east		1	2
15 `karel.pickThing();`						
	1	2	east	parcel	1	2
16 `karel.move();`						
	1	3	east	parcel	1	3
17 `karel.turnLeft();`						
	1	3	north	parcel	1	3
18 `karel.turnLeft();`						
	1	3	west	parcel	1	3
19 `karel.turnLeft();`						
	1	3	south	parcel	1	3

	karel				parcel	
Program Statement	**street**	**avenue**	**direction**	**backpack**	**street**	**avenue**
20 `karel.move();`						
	2	3	south	parcel	2	3
21 `karel.putThing();`						
	2	3	south		2	3
22 `karel.move();`						
	3	3	south		2	3

(table 1-2) ***continued***

A table recording the change of program state while tracing the execution of the `DeliverParcel` *program in Listing 1-1*

Like the state change diagram, the table is organized to show the state of the program both before and after each statement is executed. It does this by inserting the statements between the rows that record the program's state.

The first row of the table gives the initial state as established when the objects are constructed in lines 8–10. This state reflects the initial situation shown in Figure 1-10. As we trace the program, we list the statement executed and then the resulting state. The effect is equivalent to a long series of state change diagrams like the one in Figure 1-12, but considerably easier to manage.

After tracing the program, we see that the robot finishes on intersection (3, 3) as the final situation requires. In addition, it has picked up the thing and deposited it on intersection (2, 3).

Tracing the program helps us understand what it does and increases our confidence in the correctness of the solution. As you trace a program, you must do exactly what the program says. It is tempting to take shortcuts, updating the table with what we intend the program to do. The computer, however, does not understand our intentions. It does exactly what the program says. If we don't do the same while tracing, the value of tracing is lost and we can no longer claim confidence in the correctness of the solution.

KEY IDEA

Computers follow the program exactly. When tracing the execution, we must also be exact.

Having performed a trace, we now understand that the sequence of statements in the program is important. If lines 14 and 15 are reversed, for instance, the robot would try to pick up the thing before it arrives at the thing's intersection. The result would be a broken robot on intersection (1, 1).

Sequential execution is a fundamental pattern in how we solve problems. We often give directions that follow the form "do _____, and then _____": "go to the stoplight and then turn right" or "add the eggs and then beat the batter for two minutes." The *and then* indicates sequential execution.

PATTERN

Sequential Execution

1.4.6 Another Example Program

A second example program is shown in Listing 1-2. Comparing it with the `DeliverParcel` program in Listing 1-1 reveals many common elements—and where the differences are. The initial and final situations are shown in Figure 1-13. In this program, a robot (`mark`) must move around a roadblock to meet a friend (`ann`) on intersection (2, 1). This program is interesting because it uses multiple objects that belong to the same class (two robot objects and two wall objects).

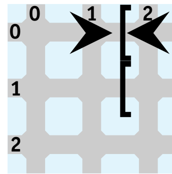
One obvious difference between this program and `DeliverParcel` is the use of `Wall` objects. A wall is instantiated using the same pattern as `Robot` and `City` objects, as shown in the following statement:

```
Wall blockAve0 = new Wall(ny, 0, 2, Direction.WEST);
```

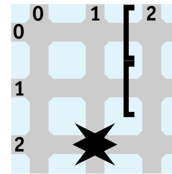
This statement constructs a wall on the western boundary of the intersection at (0, 2). This is the wall that prevents `mark` from proceeding west from its current location. The object is referenced with the name "`blockAve0`," a name chosen by the programmer.

(figure 1-13)

Initial and final situations for two robots, `mark` *and* `ann`



In the initial situation, two robots, `mark` and `ann`, are on opposite sides of a roadblock. They would like to meet on intersection (2, 1).



The final situation, where `mark` and `ann` have gone around the roadblock to meet on intersection (2, 1).

You may want to stop here and think about how you would write a program to solve this problem. What changes would you make to the `DeliverParcel` program? Then look at Listing 1-2 and see how much of it makes sense to you.

`GoAroundRoadBlock` and `DeliverParcel` have many features in common. If both programs were written on transparencies and superimposed, they would be identical in a number of places. In particular, the first six lines are nearly identical, with the exception of one name chosen by the programmers. In addition, both programs have two closing braces at the end and the organization of the code in the inner most set of braces is similar—first objects are declared and then messages are sent to them. Both programs have similar patterns of constructing objects, obtaining services from those objects, and requiring statements to be in a particular sequence.

ch01/roadblock/

Listing 1-2: *A program where* `mark` *goes around a road block and meets* `ann`

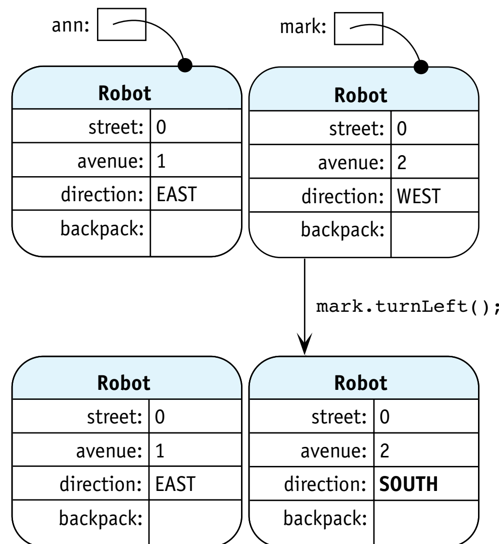
```
1  import becker.robots.*;
2
3  public class GoAroundRoadBlock
4  {
5    public static void main(String[] args)
6    {
7      // Set up the initial situation
8      City  ny        = new City();
9      Wall  blockAve0 = new Wall(ny, 0, 2, Direction.WEST);
10     Wall  blockAve1 = new Wall(ny, 1, 2, Direction.WEST);
11     Robot mark      = new Robot(ny, 0, 2, Direction.WEST);
12     Robot ann       = new Robot(ny, 0, 1, Direction.EAST);
13
14     // mark goes around the roadblock
15     mark.turnLeft();
16     mark.move();
17     mark.move();
18     mark.turnLeft();       // start turning right as three turns left
19     mark.turnLeft();
20     mark.turnLeft();       // finished turning right
21     mark.move();
22
23     // ann goes to meet mark
24     ann.turnLeft();        // start turning right as three turns left
25     ann.turnLeft();
26     ann.turnLeft();        // finished turning right
27     ann.move();
28     ann.move();
29     ann.turnLeft();
30   }
31 }
```

Object Instantiation

Command Invocation
Sequential Execution

Multiple Objects

The `GoAroundRoadBlock` program contains two robots, `mark` and `ann`. Each has its own internal state, which is completely independent of the other. `mark` can turn and move without affecting `ann`; `ann` can turn and move without affecting `mark`. The state change diagram in Figure 1-14 shows an object for `mark` and another for `ann`. After `mark` turns, `ann`'s object has not changed (although `mark`'s direction is now different).



(figure 1-14)

State change diagram illustrating the independence of two robot objects

KEY IDEA

The only way to change an object's attributes should be via its services.

In a well-designed object-oriented program, an object's state does not change unless a message has been sent to the object. When `mark` is sent the `turnLeft` message, no message is sent to `ann`, so that object doesn't change. This property of not changing unless an explicit message is received is called **encapsulation**. An object builds a capsule or wall around its attributes so that only the object's services can change them. One strength of object-oriented programming is that it makes encapsulation easy. Earlier programming methodologies did not make encapsulation as easy, with the result that programmers were often left wondering "Now, how did that information get changed?"

It is possible to write Java programs that do not use encapsulation, but it is not recommended.

1.4.7 The Form of a Java Program

Let's turn now to the first six lines and the last two lines of the `DeliverParcel` and `GoAroundRoadBlock` programs. Both programs are almost identical in these areas; we take Listing 1-1 as our example. For programs involving robots, only one of these eight lines vary from program to program. The statements we are interested in are shown in Listing 1-3, with the part that changes shown in a box.

Listing 1-3: *The "housekeeping" statements in a Java program that simulate robots*

```
 1  import becker.robots.*;
 2
 3  public class DeliverParcel
 4  {
 5    public static void main(String[] args)
 6    {
        // lines 7-22 omitted
23    }
24  }
```

The programmer chooses this name for each program.

Java Program

Line 1 makes a large body of code written by other people easily available. The 24 lines of code contained in Listing 1-1 are not nearly enough to specify everything that this program does. The program makes use of more than 3,700 lines of code written by the textbook's author. That, in turn, uses many tens or even hundreds of thousands of lines of code written by still other programmers.

All that code is organized into **packages**. The code written by the author is in the package `becker.robots`. It is a group of about 50 classes that work together to enable robot programs. Line 1 makes those classes more easily accessible within the `DeliverParcel` program.

LOOKING AHEAD

In Section 1.6, we will see programs that import packages for manipulating a window on the screen.

The class name on line 3 is simply that—a name by which this class will be known. The name of the file containing the classes' source code must be the same as this name, with "`.java`" added to the end (for example, "`DeliverParcel.java`").

The braces on lines 4 and 24 enclose the statements that are specific to this class.

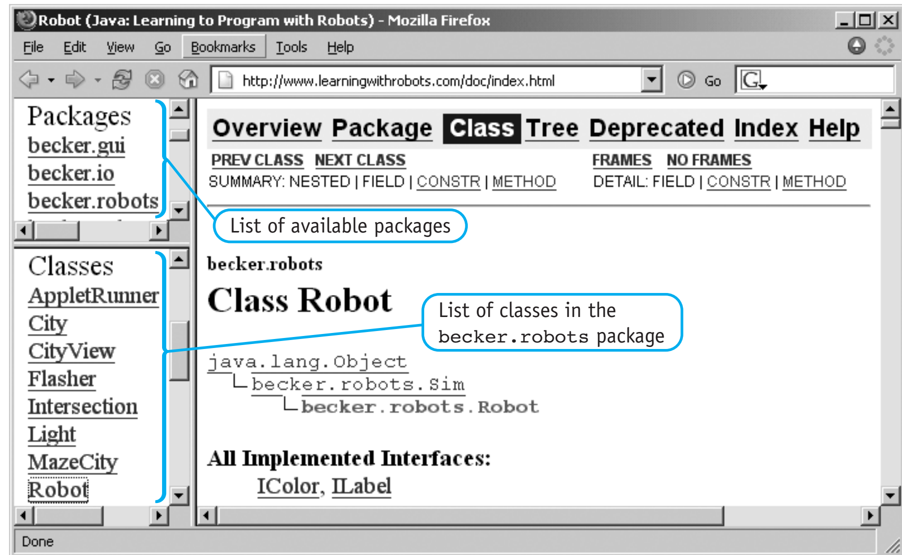
The special name `main` appears in every Java program. It marks where execution of the program begins. The words surrounding `main` are required and tell Java more about this code. The braces in lines 6 and 23 enclose all of the statements associated with `main`.

1.4.8 Reading Documentation to Learn More

The `Robot` class includes more than 30 services—too many to discuss here. Furthermore, the software accompanying this book contains more than 100 other classes. How can you find out about the other classes and all the services they (including `Robot`) offer?

The creators of Java have included a facility to extract information from the Java source code and put it in a form suitable for the World Wide Web. A sample Web page

documenting the `Robot` class is shown in Figure 1-15. You can use a Web browser to find it at *www.learningwithrobots.com* or look in the documentation directory of the CD that accompanies this book.



(figure 1-15)

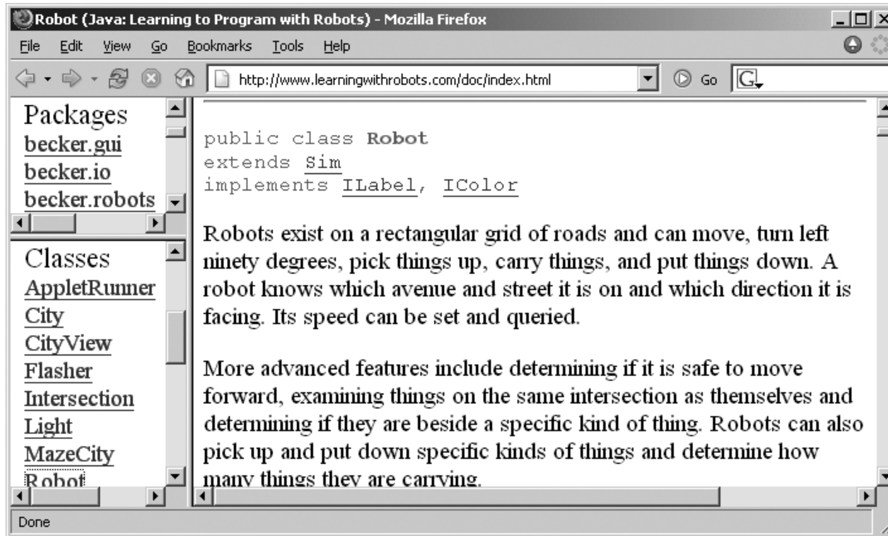
Web page showing a list of the available packages, classes within the `becker.robots` *package, and some of the documentation for the* `Robot` *class*

The upper-left panel in the window shows an area labeled “Packages”. It contains `becker.robots` and several other packages. Clicking one of these links displays in the lower-left panel a list of the classes contained in that package. For example, if you click `becker.robots`, the lower-left panel displays the classes contained in the `becker.robots` package. The classes used in our programs thus far (`City`, `Robot`, `Wall`, and `Thing`) are listed here.

Clicking one of the class names displays documentation for that class in the main part of the window. For example, if you click the `Robot` class, its documentation appears, the beginning of which is shown in Figure 1-15. It shows the relationship between the `Robot` class and a number of other classes. We’ll learn more about these relationships in Chapter 2.

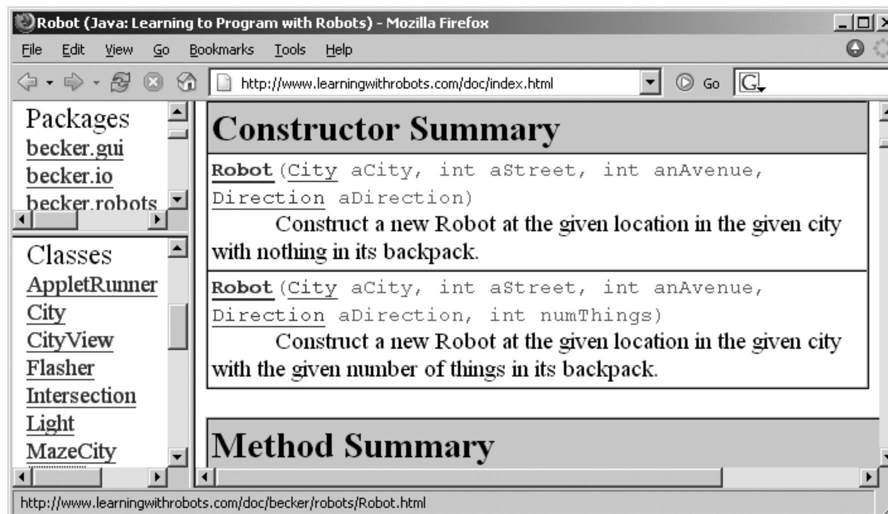
Figure 1-16 shows the documentation’s descriptive overview of the `Robot` class. The overview is used to describe the purpose of the class and sometimes includes sample code.

The documentation includes a summary description of each constructor and service (called methods in the documentation). Figure 1-17 shows the summaries for two constructors, one of which hasn’t been mentioned in this textbook. Finally, the documentation also contains detailed descriptions of each constructor and service. Figure 1-18 shows the detailed description for the `move` service.



(figure 1-16)

Descriptive overview of the `Robot` *class*

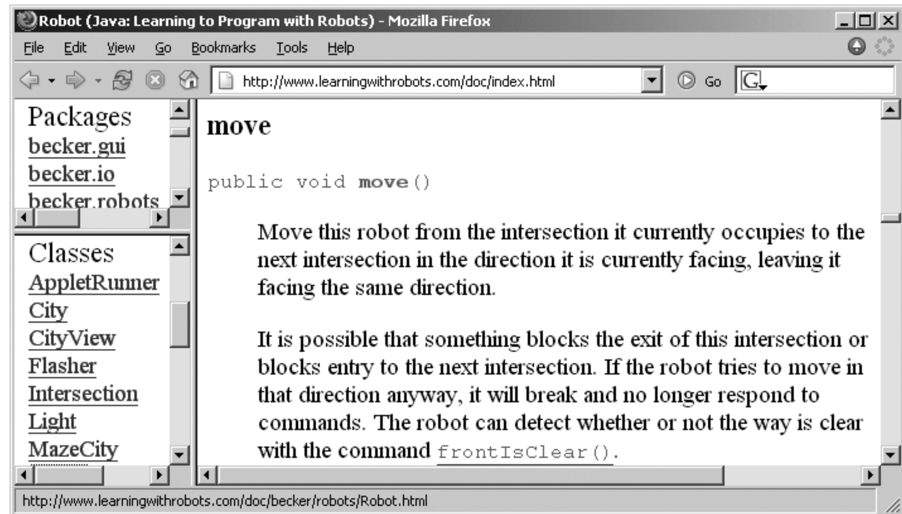


(figure 1-17)

Summary descriptions of `Robot` *constructors; each method has a similar summary*

(figure 1-18)

Detailed description of the `move` *service in the* `Robot` *documentation*

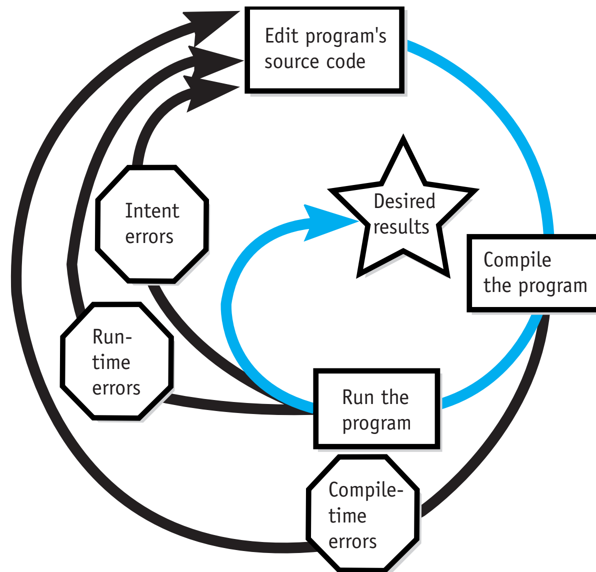


1.5 Compiling and Executing Programs

Now that we have seen two programs and discussed the overall form of a Java program, it is time to discover how to run them on a computer to see them in action. The exact steps to follow vary from computer to computer, depending on the operating system (Unix, Windows, Macintosh OS, and so on), which software is being used, and how that software has been installed. In broad strokes, however, there are three steps:

- Edit the source code in an appropriately named file with a text editor such as `vi` (Unix), `Notepad` (Windows), or `TextEdit` (Macintosh), or an integrated development environment (IDE) such as `JCreator` or `jGrasp`.
- Translate the program from the Java source code into an internal representation called **byte code** which is more easily understood by the computer. This process is called **compiling** and is performed by a program called a **compiler**.
- Run the compiled program.

Errors are often revealed by one of the last two steps. You must then go back to the first step, make changes, and try again. These steps result in a cycle, illustrated in Figure 1-19, of editing, compiling, and running, which is repeated until the program is finished. When things go right, the light path is followed, providing the desired results. When things go wrong, one of the three dark paths is taken and the source code must be edited again to fix the error.



(figure 1-19)

The edit-compile-run cycle

What kinds of things can go wrong? There are three kinds of errors: compile-time errors, run-time errors, and intent errors. **Compile-time errors** are exposed when the compiler translates the program into byte code. **Run-time errors** are discovered when the program runs, but attempts to do something illegal. **Intent errors** are also called **logic errors**. They occur when the program does not carry out the programmer's intentions.

1.5.1 Compile-Time Errors

Compile-time errors occur when the program does not conform to the rules of the Java language. People can easily recognize (or even overlook) errors in written English. Computers cannot. The programmer must obey the rules exactly.

Listing 1-4 contains a program with many compile-time errors. Each is explained by the boxed annotation beside it. You can find the code in `ch01/compileErrors/`.

KEY IDEA

The compiler can help you find compile-time errors.

Every time the compiler finds an error it prints an error message. These messages are helpful in finding the errors—be sure to read them carefully and use them. For example, misplaced semicolons can be a struggle for beginning programmers. The missing semicolon at line 8 results in an error message similar to the following:

```
CompileErrors.java:8: ';' expected
   Robot karel = new Robot(london, 1, 1, Direction.EAST)
                                                        ^
```

This message has three parts. The first is where the error was found. "`CompileErrors.java`" is the name of the file containing the error. It won't be long

before our programs are large enough to be organized into several files; you need to know which one contains the error. It is followed by the line number where the error was found, 8.

Second, "`';' expected`" indicates what the compiler identifies as the error.

Third is the line from the program where the compiler found the error. Beneath it is a caret (^) symbol showing where in the line the error was detected.

FIND THE CODE

ch01/compileErrors/

Listing 1-4: *A program with compile-time errors*

```
1  import becker.robots;
2
3  publik class CompileErrors
4  {
5    public static void main(String[] args)
6      karel.move();
7      City london = new Cit y();
8      Robot karel = new Robot(1ondon, 1, 1, Direction.EAST)
9
10     karal.move();
11     karel.mvoe();
12     karel.turnRight();
13     karel.turnleft();
14     move();
15     karel.move;
16   }
17   karel.move();
18 }
```


If you use an integrated development environment (IDE), it may show errors in a different format. It may even move the cursor to the line containing the error. No matter what the format is, however, you should still be able to learn the nature and location of the error.

Sometimes the messages are more cryptic than the one shown in the `CompileErrors` example, and occasionally the location of the error is wrong. For example, the compiler reports many errors if you misspell the variable name at line 8 where it is declared, but spell it correctly everywhere else. Unfortunately, none of the errors are at line 8. In other words, one error can cause many error messages, all pointing to the wrong location.

Because one error can cause many error messages, a reasonable debugging strategy is to perform the following tasks:

- Compile the program to produce a list of errors.
- Fix the most obvious errors, beginning with the first error reported.
- Compile the program again to obtain a revised list of the remaining errors.

Furthermore, do these tasks early in your program's development, and do them often. Waiting too long to compile your program will often result in many cryptic error messages that are hard to understand. Errors are easier to find and fix when you compile early and often.

KEY IDEA

Compiling early and often makes finding compile-time errors easier.

1.5.2 Run-Time Errors

Run-time errors are discovered when the program is actually run or traced. They are the result of instructions executing in an illegal context. For example, the instruction `karel.move();` will compile correctly (as long as a robot named `karel` has been constructed). However, if `karel` is facing a wall when this instruction executes, it will break. The instruction is executed in an illegal context.

The error of crashing a robot into a wall is reported in two different ways. First, the robot's icon is changed to show that the robot is broken. Second, an informative error message is printed. An example is shown in Figure 1-20.

```
1 Exception in thread "main" becker.robots.RobotException: A
  robot at (1, 2) crashed into a wall while moving WEST.
2         at becker.robots.Robot.breakRobot(Robot.java:558)
3         at becker.robots.Robot.move(Robot.java:148)
4         at GoAroundRoadBlock.main(GoAroundRoadBlock.java:17)
```

(figure 1-20)

Error message generated at run-time

This error message results from removing line 17 from Listing 1-2. The result is that `mark` does not move far enough to go around the roadblock and crashes into it.

The first line of the message contains technical information until the colon (`:`) character. A description of what went wrong usually appears after the colon. In this case, we are told that a robot crashed while moving west.

Lines 2-4 say where in the source code the error occurred and how the program came to be executing that code. Line 2 says the error happened while the program was executing line 558 in the source file `Robot.java`. That might be helpful information if we had access to `Robot.java`, but we don't. It is part of the `becker.robots` package imported into our robot programs. Line 3 indicates the error is also related, somehow, to line 148 in that same source file. Finally, line 4 mentions `GoAroundRoadBlock.java`, the file shown

in Listing 1-2. At line 17 we find a `move` instruction, the one that caused the robot to crash. This is where our efforts to fix the program should start.

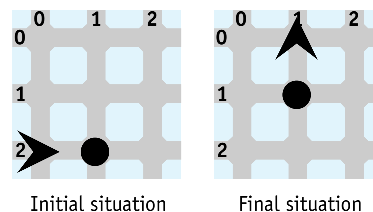
One situation that can be confusing for beginning programmers is when the Java system cannot even begin running your program. Usually, the run-time error message will contain the "word" `NoClassDefFoundError`. This means that the Java system cannot find your program to run, perhaps because it has not been successfully compiled or perhaps because the compiler did not place the compiled version of your program in the expected place.

1.5.3 Intent Errors

An intent error occurs when the program fails to carry out its intended purpose. The program may not have any compile-time or run-time errors, yet still fail to accomplish the job for which it was written. An intent error is also called a logic error.

These errors can be among the hardest to find. The computer can help find compile-time and run-time errors because it knows something is wrong. With an intent error, however, the computer cannot tell that something is wrong, and therefore provides no help other than executing the program. Remember, a computer does what it is told to do, which might be different from what it is meant to do.

For example, consider a program intended to move a thing from (2, 1) to (1, 1). The initial and final situations are shown in Figure 1-21.



(figure 1-21)

Initial and final situations for a task to move a thing

Sequential Execution

Suppose the programmer omitted the `turnLeft` instruction, as in the following program fragment:

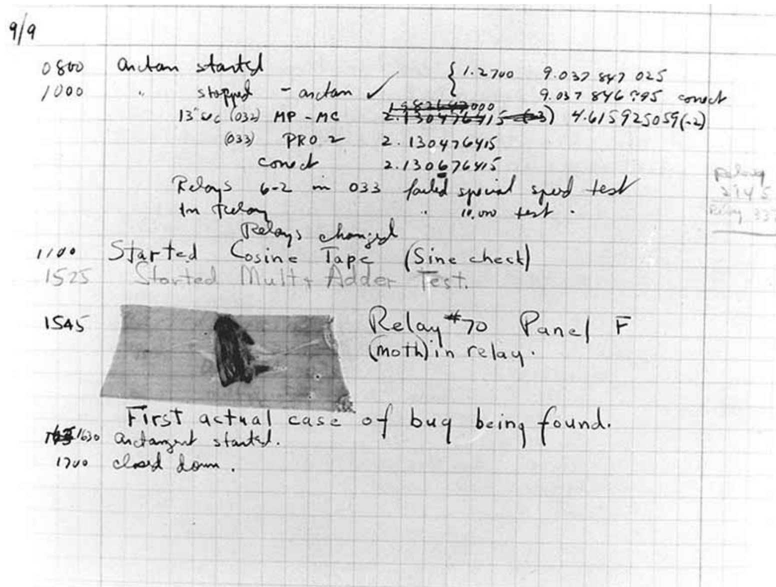
```
katrina.move();
katrina.pickThing();
                        // should have turned left here
katrina.move();
katrina.putThing();
katrina.move();
```

As a result, the robot would finish the task at (2, 3) instead of (0, 1), and the thing would be at (2, 2) instead of (1, 1)—not what was intended.

Fortunately, the visual nature of robot programs makes many intent errors easy to find. Debugging intent errors in less visual programs is usually much harder.

1.5.4 A Brief History of Bugs and Debugging

Programming errors of any kind are often called **bugs**. The process of finding and fixing the errors is called **debugging**. The origin of the term is not certain, but it is known that Thomas Edison talked about bugs in electrical circuits in the 1870s. In 1947, an actual bug (a moth) was found in one of the electrical circuits of the Mark II computer at Harvard University, causing errors in its computations (see Figure 1-22). When the moth was found, it was taped into the operator's log book with the notation that it is the "first actual case of a bug being found." Apparently, the term was already in use for non-insect causes of computer malfunctions.



(figure 1-22)

A 1947 entry from a log book for the Mark II computer at Harvard University

1.6 GUI: Creating a Window

We have learned a lot of Java programming in the context of `Robot` objects. These concepts include:

- A class, such as `Robot` or `Thing`, is like a factory for making as many objects as you want. Each class or factory only makes one kind of object.
- A new object is instantiated with the `new` operator, for instance `Robot mark = new Robot(ny, 0, 2, Direction.WEST);`.

- All objects belonging to the same class have the same services, but each has its own attribute values that are independent of all other objects.
- A client can invoke an object's services with the object's name, a dot, and then the name of the desired service.

These concepts are not only for robot programs, but apply to every object-oriented Java program ever written. To illustrate, each chapter of this book includes a section applying the concepts learned using robots to graphical user interfaces, or GUIs (pronounced "gooey"). Applying the concepts to classes supplied with the Java language that have nothing to do with robots shows you how these concepts can be used in many other contexts.

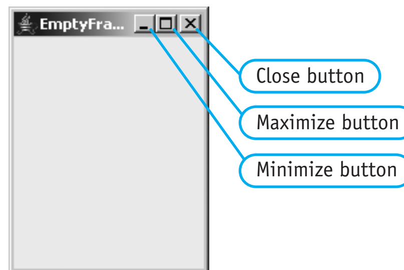
Graphical user interfaces are the part of the program that interacts with the human user. It probably obtains input from the user and displays results to the user. In terms of what the user sees, the GUI consists of the windows, dialog boxes, lists of items to select, and so on.

1.6.1 Displaying a Frame

GUIs add a lot of complexity and development time to a program. Fortunately, Java provides a rich set of resources to help develop interfaces. The beginning point is the window, called a **frame** in Java. The simplest possible frame is shown in Figure 1-23.

(figure 1-23)

Simplest Java frame



Though it is empty, the frame nevertheless has substantial functionality. If the user clicks the close box, the program quits. If the user clicks the minimize box the frame becomes as small as possible, while clicking the maximize box enlarges the frame to take up the entire screen. The user may also adjust the size of the frame by clicking and dragging an edge of the frame.

The program that displays this frame is even simpler than a robot program and is shown in Listing 1-5. Notice the similarity to Listing 1-1, including the following features:

- Both programs include `import` statements (although the actual packages differ), the declaration of the class at line 3 (although the class name differs), the placement of braces, and the declaration of the special service `main`.

- Both programs instantiate an object.
- Both programs invoke the services of an object.

Listing 1-5 *The* `EmptyFrame` *program displays an empty window*

FIND THE CODE

ch01/emptyFrame/

```
1  import javax.swing.*;      // use JFrame
2
3  public class EmptyFrame
4  {
5    public static void main(String[] args)
6    { // declare the object
7      JFrame frame = new JFrame();
8
9      // invoke its services
10     frame.setTitle("EmptyFrame");
11     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12     frame.setLocation(250, 100);
13     frame.setSize(150, 200);
14     frame.setVisible(true);
15   }
16 }
```

PATTERN

Java Program
Object Instantiation

PATTERN

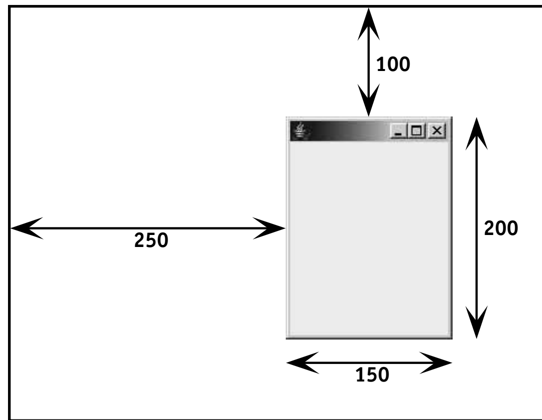
Command Invocation
Sequential Execution

The `EmptyFrame` program differs from the `DeliverParcel` program in the kinds of objects created and the services demanded of them. A `JFrame` object serves as a container for information displayed to the user. By default, however, the frame has no title, is not visible on the screen, has no area for displaying information, and hides the frame when the close box is clicked (leaving us with no good way to stop the program). The services invoked in lines 10-14 override those defaults to provide the functionality we need.

The `setTitle` service causes its argument to be displayed at the top of the frame.

The `setDefaultCloseOperation` service specifies what the frame object should do when the close box is clicked. `JFrame.EXIT_ON_CLOSE` gives a meaningful name to a particular value; `Direction.EAST` serves a similar function for robot programmers.

The `setLocation` service says where the frame should appear. Its first argument is the distance from the left side of the screen to the left side of the frame. The second argument specifies the distance from the top of the screen to the top of the frame. The `setSize` service specifies the size of the frame. The first argument is the width and the second argument is the height. All of the arguments to these services are given in **pixels**, an abbreviation for **picture elements**, which are the tiny dots on the screen that make up the image. The meaning of these arguments is illustrated in Figure 1-24.




```
JFrame frame = new JFrame();
...
frame.setLocation(250, 100);
frame.setSize(150, 200);
```

(figure 1-24)

Relationship of the arguments to `setLocation` *and* `setSize` *to the frame's location and size; the outer rectangle represents the computer monitor*

LOOKING AHEAD

`true` *and* `false` *are Boolean values used to represent true or false answers, and are covered in detail in Chapter 4.*

Finally, the `setVisible` service specifies whether the frame is visible on the screen. If the argument is `true`, the frame will be visible, while a value of `false` will hide it.

1.6.2 Adding User Interface Components

A frame with nothing in it is rather boring and useless. We can fix that by setting the **content pane**, the part of a frame designed to display information. We can add to the content pane buttons, textboxes, labels, and other user interface elements familiar to modern computer users. These buttons, labels, and so on are often called **components**. A component is nothing more than an object designed to be part of a graphical user interface.

An early warning, however: The resulting programs may look like they do something useful, but they won't. We are still a long way from writing a graphical user interface that actually accepts input from the user. The following programs emphasize the *graphical* rather than the *interface*.

Figure 1-25 shows a snapshot of a running program that has a button and a text area displayed in a frame. The frame's content pane has been set to hold a `JPanel` object. The `JPanel`, in turn, holds the button and text area. Listing 1-6 shows the source code for the program.

(figure 1-25)

Frame with a content pane containing a button and a text area; the user typed, "I love Java!" while the program was running

Listing 1-6 `FramePlay`, *a program to display a frame containing a button and a text area*

FIND THE CODE

ch01/framePlay/

```
1  import javax.swing.*;      // use JFrame, JPanel, JButton, JTextArea
2
3  public class FramePlay
4  {
5    public static void main(String[ ] args)
6    { // declare the objects to show
7      JFrame frame = new JFrame();
8      JPanel contents = new JPanel();
9      JButton saveButton = new JButton("Save");
10     JTextArea textDisplay = new JTextArea(5, 10);
11
12     // set up the contents
13     contents.add(saveButton);
14     contents.add(textDisplay);
15
16     // set the frame's contents to display the panel
17     frame.setContentPane(contents);
18
19     // set up and show the frame
20     frame.setTitle("FramePlay");
21     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22     frame.setLocation(250, 100);
23     frame.setSize(150, 200);
24     frame.setVisible(true);
25   }
26 }
```

PATTERN

Display a Frame

In lines 7–10, we declare and instantiate several objects. The `JPanel` instance named `contents` is simply a container. It will hold the things we are really interested in, the button and text area. The button and text area are instances of `JButton` and `JTextArea`, respectively. The `JPanel`'s `add` service in lines 13 and 14 adds them to its list of things to display.

When the button is constructed in line 9, `"Save"` is passed to the constructor's parameter. This text appears on the button when it is displayed. When the text area is constructed in line 10, the number of lines of text it should hold and how many characters wide it should be are passed as arguments. Both measures are approximate.

You can learn much more about these classes by browsing the online documentation at *java.sun.com/j2se/1.5.0/docs/api/*.

If you run the `FramePlay` program, you will find that the objects used have quite a bit of built-in functionality. The frame responds to the close, minimize, and maximize boxes, and resizes when you drag an edge. The save button flashes when clicked, and you can type in the textbox. This amount of functionality is remarkable for a 26-line program. As with the robot programs, much of this functionality is due to the programmers who wrote the classes we are using.

KEY IDEA

The ideas you learn with robots—such as classes, objects, and services—apply to all object-oriented programs.

Again, notice the similarity between the `FramePlay` program and all the other programs in this chapter. The concepts we learned with the robot programs truly are general and can be used in all Java programs. In fact, many of the ideas apply to all programs, whether or not they are written in Java. Some ideas, such as modeling, abstraction, and patterns apply to lots of different problems, whether or not they have a computer solution. You are learning a portable set of skills that can be applied in many circumstances.

1.7 Patterns

Many patterns appear in software—problems that appear repeatedly that have the same solution. A number of these have already been identified for you in the margins of this text. Figure 1-26 shows the icon used to identify a pattern. Expert software developers know many patterns and apply an appropriate one to solve the issue at hand, almost without thinking about it. Much of your work, as you learn to program, will involve learning to recognize which software patterns apply to the issue you are facing.

(figure 1-26)

Icon used to identify an example of a pattern

Software patterns began as a way to capture and discuss big ideas, such as how to structure many classes and objects to model a particular kind of problem. In this book we extend the idea of patterns to smaller ideas that may cover only one or two lines of code. As beginning programmers, our attention will be focused primarily on these elementary patterns.

Our elementary patterns have five elements: name, context, solution, consequences, and related patterns. In the pattern expositions they are clearly shown by name and typography, as shown in Figure 1-27. However, instead of describing an actual pattern, the figure describes what each section of a pattern involves.

Name: The name gives programmers a common vocabulary with which to discuss their work. Soon you will be able to say to a classmate, "I think the Command Invocation pattern applies here," and your classmate will know the kind of issue you think needs solving, your proposed solution, and the consequences of implementing that solution. Naming concepts increases our ability to work with abstractions and communicate them to others.

Context: The context describes the situations in which the pattern is applicable. Obviously, the pattern's context must match the context of your programming issue for the pattern to be useful.

Solution: The solution describes an approach to resolving the programming issue. For many patterns, the most appropriate form for the solution is several lines of code, likely with well-defined places where the code must be customized for the particular issue at hand. The places to customize appear in italics between « *and* ». For other patterns, an appropriate form for the solution is a class diagram that shows how two or more classes work together to resolve the issue.

Consequences: The consequences describe the natural by-products of applying this particular pattern. Sometimes these consequences are good, and sometimes they are bad; sometimes they are some of both. Weighing the consequences is part of deciding whether this pattern is the one to apply to your issue. It may be that another pattern can be applied in the same context with a better result.

Related Patterns: Finally, related patterns name patterns that have a different approach to resolving the issue or are based on the same ideas.

(figure 1-27)

Parts and format of a pattern description

LOOKING AHEAD

Naming is a powerful idea. Here we are naming patterns. In the next chapter, we will name a sequence of instructions. Doing so increases our power to think about complex problems.

Patterns are found or discovered, not invented. They result from sensing that you are repeating something you did earlier, investigating it enough to find the common elements, and then documenting it for yourself and others. Because patterns arise from experience, they are a good way to help inexperienced programmers gain expertise from experienced programmers.

The patterns listed here should feel familiar. You have seen them a number of times already, often accompanied with a margin note like the one shown here. Once the pattern has been formally presented, however, we will no longer call attention to its application with margin notes.

PATTERN

Command Invocation

1.7.1 The Java Program Pattern

Name: Java Program

Context: You want to write a Java program.

Solution: Implement a class that contains a service named main. In particular, customize the following template:

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

public class «programClassName»
{
  public static void main(String[] args)
  { «list of statements to be executed»
  }
}
```

The import statement makes classes from the named package easier to access. Typical values for *«importedPackage»* include becker.robots.* for robot programs, and java.awt.* and javax.swing.* for programs with graphical user interfaces.

The *«className»* is chosen by the programmer and must match the filename containing the source code.

The *«list of statements to be executed»* may include statements following the Object Instantiation pattern and Command Invocation patterns, among others.

Consequences: A class is defined that can be used to begin execution of the program.

Related Patterns: All of the other patterns in this chapter occur within the context of the Java Program pattern.

1.7.2 The Object Instantiation Pattern

Name: Object Instantiation

Context: A client needs to instantiate or construct an object to carry out various services for it.

Solution: Instantiate the object using the new keyword and a constructor from the appropriate class. Provide arguments for all of the constructor's parameters. Finally, assign the object reference provided by new to a variable. Examples:

```
City manila = new City();
Robot karel = new Robot(manila, 5, 3, Direction.EAST);
JButton saveButton = new JButton("save");
```

In general, a new object is instantiated with a statement matching the following template:

```
«variableType» «variableName» =
           new «className»(«argumentList»);
```

The «*variableName*» is used in the Command Invocation pattern whenever services must be carried out by the object.

Until we have studied polymorphism in Chapter 12, «*variableType*» and «*className*» will be the same. After that, they will often be different, but related. The «*className*», of course, determines what kind of object is constructed. The «*variableName*» should reveal the purpose or intent of what it names.

LOOKING AHEAD

We will say much more about choosing variable names wisely in Section 2.4.2.

Consequences: A new object is constructed and assigned to the given variable.

Related Patterns: The Command Invocation pattern requires this pattern to construct the objects it uses.

1.7.3 The Command Invocation Pattern

Name: Command Invocation

Context: A client wants an object to perform one of the services it provides.

Solution: Provide a reference to the object, a dot, the name of the desired service, and any arguments. Information about how to use the command, including any arguments it requires, can be found in its documentation. A command invocation is always terminated with a semicolon. Examples:

LOOKING AHEAD

Queries are invoked in a different context than commands. Although they are both services, they have different invocation patterns.

```
karel.move();
collectorRobot.pickThing();
frame.setSize(150, 200);
contents.add(saveButton);
```

In general, a command is invoked with a statement matching the following template:

```
«objectReference».«commandName»(«argumentList»);
```

where «*objectReference*» is a variable name created using the Object Instantiation pattern.

Consequences: The command is performed by the named object. It is possible to invoke a command in an invalid context, resulting in a run-time error.

Related Patterns:

- The Object Instantiation pattern must always precede this pattern to create the object.
- The Sequential Execution pattern uses this pattern two or more times.

1.7.4 The Sequential Execution Pattern

Name: Sequential Execution

Context: The problem you're working on can be solved by executing a sequence of steps. The order of the steps matters, because later steps depend on earlier steps to establish the correct context for them to execute.

LOOKING AHEAD

Problem 1.4 further explores the idea of "correct order."

Solution: List the steps to be executed in a correct order. A correct order is one where each statement appears after all the statements upon which it depends. Each statement is terminated with a semicolon. An example from the `DeliverParcel` program (see Listing 1-1) demonstrates the solution:

```
 7        // Set up the initial situation
 8        City prague = new City();
 9        Thing parcel= new Thing(prague, 1, 2);
10        Robot karel = new Robot(prague, 1, 0, Direction.EAST);
11
12        // Direct the robot to the final situation
13        karel.move();
14        karel.move();
15        karel.pickThing();
```

Lines 9 and 10 require a city when they are constructed; they therefore must appear after line 8, where the city is constructed. However, lines 9 and 10 are independent of each other and can appear in either order.

The robot `karel` cannot pick up a `Thing` at (2, 1) until it has reached that intersection. Lines 13 and 14 must therefore come before the `pickThing` command in line 15 to establish the context for it to execute (that is, move the robot to the intersection containing the `Thing` it picks up).

LOOKING AHEAD

Long sequences of statements are hard to understand. In Chapter 2 we will learn methods to help keep such sequences short.

Consequences: Each statement, except the first one, may depend on statements that come before it. If any of those statements are out of order or wrong, an error or unexpected result may appear later in the program, which can make programming a tedious process.

This model of execution assumes that each statement executes completely before the next one has begun. It is as if each statement were strung on a thread. Follow the thread from the beginning and each statement is reached in order. Follow the thread back through time and you have a complete history of the statements executed prior to the current statement.

Related Patterns: The Command Invocation pattern is used two or more times by this pattern.

1.7.5 The Display a Frame Pattern

Name: Display a Frame

Context: A program must show some visual information to the user.

Solution: Organize the visual information in a `JPanel` which is displayed within a `JFrame`.

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

public class «programClassName»
{
  public static void main(String[] args)
  { // declare the objects to show
    JFrame «frame» = new JFrame();
    JPanel «contents» = new JPanel();

    «statements to declare and add components to contents»

    // set the frame's contents to display the panel
    «frame».setContentPane(«contents»);

    // set up and show the frame
    «frame».setTitle("«title»");
    «frame».setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    «frame».setLocation(«xPos», «yPos»);
    «frame».setSize(«width», «height»);
    «frame».setVisible(true);
   }
}
```

Consequences: The frame is displayed along with the contents of the `JPanel`. The `JFrame`'s functionality is automatically available, including resizing, minimizing, and closing the frame.

Related Patterns:

- This pattern is a specialized version of the Java Program pattern.
- The Model-View-Controller pattern (Chapter 13) builds further on this pattern.

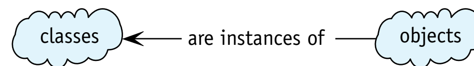
1.8 Summary and Concept Map

In this chapter we have learned that programs implement models. Object-oriented programs, such as those written in Java, use software objects to do the modeling. These software objects offer services to either perform a command or answer a query for a client. Documentation helps us learn more about the services an object offers.

Objects are instantiated from classes, a sort of template or definition for objects. Classes in the robot world include `Robot`, `City`, `Wall`, and `Thing`. When instantiating an object, we often assign it to a variable, allowing us to refer to it using the variable's name in many places in the program.

1.8.1 Concept Maps

Each chapter of this textbook includes a **concept map** as part of the summary. A concept map shows the major concepts in the chapter and relates them to each other with short phrases. The short phrases should be read in the direction of the arrow. For example, the following portion of the diagram should be read as "objects are instances of classes."

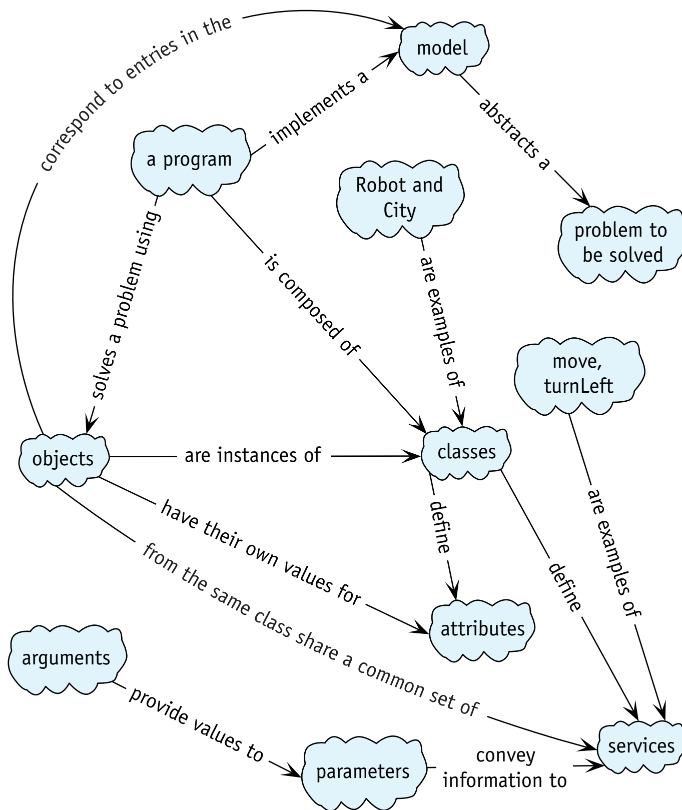


By studying the concept map, you remind yourself of important vocabulary (such as class, object, and instance in the preceding example) and the relationships between concepts.

Three suggested ways to use the concept maps as study tools are:

- After reading the chapter but before you look at the concept map, try drawing your own concept map and then compare it with the one in the chapter. They will undoubtedly be different, but hopefully will include many of the same concepts. The differences will identify places for you to clarify your understanding.
- Try reproducing the concept map but with the arrows running backwards. You will need to adjust the connecting phrase accordingly. Consider the following example:

- The concept maps include the most important concepts, but not all of them. Try to integrate additional concepts into the map. For example, how could you include the concepts of constructors, initial situations, and final situations in the following concept map?



1.9 Problem Set

Problem sets present three types of problems: written exercises, programming exercises, and programming projects. Written exercises do not require programming but may require a computer to read documentation. Programming exercises and projects both require programming, but to a different degree. Exercises are short, such as changes to existing code. Programming projects require considerably more effort.

Written Exercises

1.1 Figure 1-4 shows a class diagram for the `Concert` class. Create a similar class diagram for a `Book` class. It's part of a program at the local public library that loans books to library patrons.

1.2 Trace the following program. In a table similar to Table 1-2, record `karel`'s current street, avenue, direction, and contents of its backpack. If a run-time error occurs, describe what went wrong and at which line. There are no compile-time errors.

```
 1 import becker.robots.*;
 2
 3 public class Trace
 4 {
 5   public static void main(String[] args)
 6   { City paris = new City();
 7     Thing theThing = new Thing(paris, 1, 2);
 8     Wall w = new Wall(paris, 1, 2, Direction.WEST);
 9     Robot karel = new Robot(paris, 1, 0, Direction.EAST);
10
11     karel.move();
12     karel.turnLeft();
13     karel.turnLeft();
14     karel.turnLeft();
15     karel.move();
16     karel.turnLeft();
17     karel.move();
18     karel.turnLeft();
19     karel.move();
20     karel.turnLeft();
21     karel.pickThing();
22     karel.move();
23   }
24 }
```

1.3 Make a table similar to Table 1-2 and trace the program in Listing 1-2. You will need to add extra columns so you can record values for both robots.

1.4 The `DeliverParcel` program in Listing 1-1 makes extensive use of the Sequential Execution pattern. For many pairs of consecutive statements, interchanging the statements causes the program to fail. Give a pair of statements (that are not identical) where changing the order does *not* cause the program to fail.

1.5 In the `DeliverParcel` program in Listing 1-1, change line 8 to read `City prague = new City(5, 10);`. Based on the online documentation for `City`, what effect will this change have?

1.6 The following program contains 12 distinct compile-time errors. List at least nine of them by giving the line number and a short description of the error.

```
 1 import becker.robots;
 2
 3 Public class Errors
 4 {
 5   public static void main(String[] args)
 6   { City 2ny = new City(5, 5);
 7     Thing aThing = new Thing(2ny, 2, 1);
 8     Wall eastWall = Wall(2, 1, EAST);
 9     Robot karel = new Robot(2ny, 0, 1, Direction.EAST);
```

```
10
11       karel.move();
12       karel.move(); move();
13       karel.pickthing();
14       karel..turnLeft(); karel.turnLeft();
15       kerel.mve();
16       karel.turnLeft(3);
17       karel.move;();
18       karel.putThing();
19     }
20  }
```

1.7 The `JButton` constructor used in the `FramePlay` program in Listing 1-6 uses a string as a parameter. `JFrame` has a constructor that also has a string as a parameter. What is the effect of replacing line 7 with `JFrame frame = new JFrame("Mystery");`? Which line of the existing program should also change? How? (*Hint*: Consult the online documentation.)

Programming Exercises

1.8 Beginning with the program in Listing 1-1, deliberately introduce one compile-time error at a time. Record the compiler error generated and the cause of the error. If one error causes multiple messages, list only the first two. Find at least six distinct compile-time errors.

1.9 Write a program that creates a robot at (1, 1) that moves north five times, turns around, and returns to its starting point.

a. Run the program and describe what happens to the robot.

b. Describe a way to use the controls of the running program so you can watch the robot the entire time it is moving.

c. Consult the online documentation. Describe a way to change the program so you can watch the robot the entire time it is moving.

1.10 Make a copy of the `FramePlay` program in Listing 1-6. Add a new `JCheckBox` component.

a. Run the program and describe the behavior of the new component.

b. The `JCheckBox` constructor can take a string as a parameter, just like `JButton`. What happens if you use the string “Show All Items”?

1.11 Run the program in Listing 1-2. In your Java development software, choose Save from the File menu, enter a filename, and click the Save button. Find the file that was created.

a. Describe the contents of the file.

b. Search the documentation for the `City` class to find a use for this file. Describe the use.

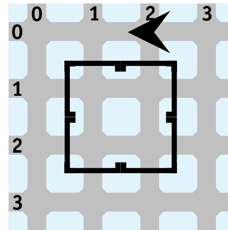
c. Modify the program to use the file that was created. (*Hint*: You will need to construct the robots yourself within `main`.) Modify the file to place additional walls and things within the city.

Programming Projects

1.12 Write a program that begins with the initial situation shown in Figure 1-28. Instruct the robot to go around the walls counter clockwise and return to its starting position. *Hint:* Setting up the initial situation requires eight walls.

(figure 1-28)

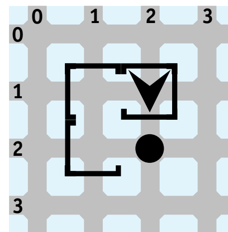
Walking around the walls



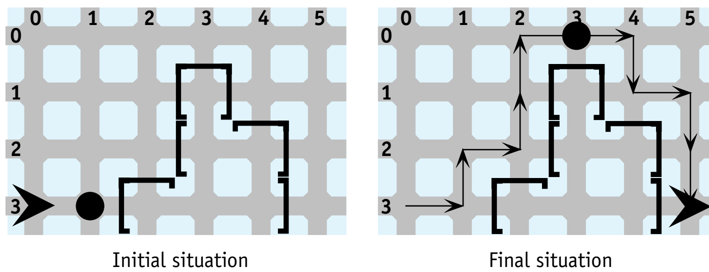
1.13 Every morning `karel` is awakened when the newspaper, represented by a `Thing`, is thrown on the front porch of its house. Instruct `karel` to retrieve the newspaper and return to "bed." The initial situation is as shown in Figure 1-29; in the final situation, `karel` is on its original intersection, facing its original direction, with the newspaper.

(figure 1-29)

Fetching the newspaper



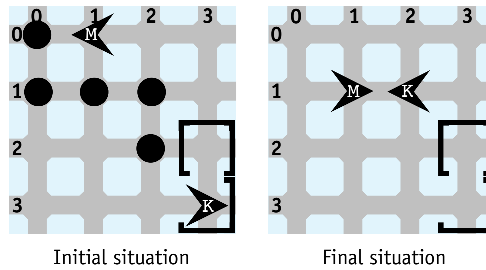
1.14 The wall sections shown in Figure 1-30 represent a mountain (north is up). Write a program that constructs the situation, and then instructs the robot to pick up a flag (a `Thing`), climb the mountain, and plant the flag at the top, descending down the other side. The robot must follow the face of the mountain as closely as possible, as shown by the path shown in the final situation.



(figure 1-30)

Initial and final situations for a robot that climbs a mountain

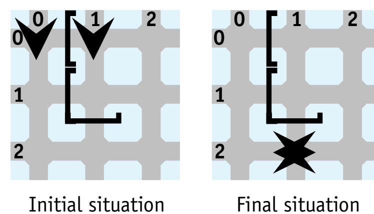
1.15 On the way home from the supermarket, `karel`'s bag rips slightly at the bottom, spilling a few expensive items (`Things`) on the ground. Fortunately, `karel`'s neighbor `maria` notices and calls to him as `karel` arrives home. This initial situation is shown on the left in Figure 1-31. Write a program in which `karel` and `maria` both begin picking up the items, meeting as shown in the final situation. Use the `setLabel` service in `Robot` to label each robot.



(figure 1-31)

Initial and final situations for robots picking up dropped items

1.16 Write a program that begins with the initial situation and ends with the final situation shown in Figure 1-32. In the final situation, the robot originally at (0, 0) is facing east and the robot originally at (0, 1) is facing west. Alternate the actions of the two robots so they arrive at their destination at approximately the same time.



(figure 1-32)

Initial and final situations for two robots moving and meeting each other

1.17 The `JTextArea` class includes the services `setText` and `append`. Read the online documentation to understand what they do and how to use them. The arguments to both might be something like, `"My second point is..."`.

a. Modify the `FramePlay` program from Listing 1-6 so that it displays a short sentence as the program runs; it is *not* typed in by the user as shown in Figure 1-25.

b. Describe what happens if you use a much longer sentence.

c. Research the `JTextArea` class. Find a way to display all the words in your longer sentence.

1.18 Modify the `FramePlay` program from Listing 1-6 to make the frame about three times as wide and twice as tall. Instead of adding a `JButton` and `JTextArea`, add a `JColorChooser` component. Run the program to answer the following questions:

LOOKING AHEAD

This program can be used to choose an appropriate color for Listing 2-6.

a. What color results from a red value of 255, a green value of 255, and a blue value of 200 (255, 255, 200)?

b. How is (255, 255, 0) different from (255, 255, 200)?

c. What color is (255, 193, 0)?

d. The RGB color model specifies the amounts of red, greens and blue that make up a color. There are other models. The HSB model, for example, specifies the hue, saturation, and brightness of a color. Specify RGB and HSB values for a brown color that pleases you. Which model is easiest for you to use? Why?