1 Objects

Controlling a Robot Butler
1.1 Objectives

Concepts

This chapter describes what it means for a programming language to be object-oriented. As a part of this process, basic computer language and object terms are defined. These terms will become the basis for communicating ideas throughout this text.

Object concepts are illustrated using robot butler simulation. This particular example is chosen because there are obvious mappings between physical objects and software objects.

Patterns

Patterns are independent of a particular programming language and are useful throughout our programming life. In this chapter patterns will be defined and example patterns will be described. We begin with elementary control patterns. The first pattern we encounter specifies how the order in which we write program statements determines the order in which they are executed. The sequential pattern is one of the most underrated.

Java Skills

Next this chapter addresses how Java can be used to express the above concepts and patterns.

- Identifying tokens, identifiers, and keywords Section 1.4, page 16
- Creating objects Section 1.5.1, page 20
- Declaring object identifiers Section 1.5.2, page 21
- Assigning objects to identifiers Section 1.5.3, page 22
- Applying operations to objects Section 1.6, page 23
- Writing comments Section 1.7, page 24

Abilities

By the end of this chapter we should be able to do the following:

Modify an existing program by adding objects and applying operations to them.

Use software tools (provided by the local institution) to edit, compile, link, and execute a Java applet.
1.2 A Robot for a Butler

On July 4, 1997, a spacecraft named Pathfinder bounced to a halt on the surface of Mars. After deflating the air bags that cushioned its fall, a small rover named Sojourner began its odyssey. Mars is a long way from Earth. It takes 11 minutes for commands to reach Pathfinder and then they are relayed to Sojourner, which acknowledges them to Pathfinder. Next it takes another 11 minutes for their impact to be seen on Earth. This 22 minute plus delay creates problems for scientists on Earth; a problem made more difficult because communication can be broken by events on Earth or the passage of Sojourner behind a rock. The Sojourner design team solved these problems by making Sojourner work autonomously.

Robots are machines that link stimulus to actions. They use devices called sensors to check their surroundings and actuators to manipulate the world.

Sojourner is an interesting collaboration of humans and machines. Scientists in Pasadena, California, plot a day’s journey for Sojourner and send it up. Sojourner executes the plan several hours later. These plans tell Sojourner to move from point A to point B, but not how to get there. It is Sojourner’s job to use an internal set of rules to pick a path around small obstacles while eventually heading toward its goal. Sojourner’s design is an example of the current status of robotics. Robots can create plans for simple goals, but managing an overall exploration plan is not yet practical.

Robots have applications on Earth as well. They are used in manufacturing and for handling hazardous waste. All of us can anticipate the benefits of a robot servant designed to make our life easier. It would be great if each of us had a robot assistant to manage some of the mundane tasks in our life, such as doing the laundry, cleaning up our rooms, fixing meals, and running errands. In this chapter we will program a machine to pick up the laundry in our room. Unlike Sojourner, which can plan how to solve a simple task, we will program explicitly every move of our robot assistant.
1 Objects

1.2.1 Objects and Classes

Computers can keep track of medical records, play games, and guide airliners. In some cases software may be life-critical. Object-oriented (OO) software development is one of many methodologies that have been developed to help us write high-quality and cost-effective software.

Suppose we are given the task of creating a robot butler that will pick up the dirty clothes in our room. We will need to instruct our butler to move forward, turn, and pick up clothes. There are several references to objects and operations in this task. For example, butler is a class of objects that has specific physical characteristics and a set of behaviors shared by all objects of the butler class. All butlers have arms for carrying objects, legs for moving, and sensors for detecting objects; and all respond to a set of commands.

Objects have an identity, a state, and operations. Objects have an identity because we must have a means to refer to them. In many cases we refer to an object through its name. For example, the name James may identify a particular butler. Some objects do not have names but we can still refer to them indirectly. For example, we may refer to a pencil as the one in our backpack or we may refer to a coat as the one on the third hook.

Objects have a state. The state of an object indicates its current condition. For example, the state of a butler includes its location, the direction it is facing, and whether it is carrying anything. We may refer to the state of our car as being “out of gas” or the state of our little sister as being cranky. When we design software objects, we choose items of state that are important for the problem at hand. In the case of a butler robot in this chapter’s program, a butler’s location is important, so location is made part of its state. Its color and temperature are not important to our simulation, so they are not included as part of the state. (In other applications—for example, a model of heat dissipation in robots—color and temperature are important and would be part of a butler’s state.) One of the tasks of a program designer is to choose the critical state attributes of an object.

Objects have operations. Operations change the state or report on the state of an object. Operations on a butler robot include move forward, turn, and pick up an object. For example, moving forward changes the state of our robot by giving it a new location. Turning changes the state of our robot by giving it a new direction.

Objects have an identity, a state, and operations. We refer to an object by its identity. An object’s state describes its current conditions. Operations are actions that change or report on an object’s state.
Objects are instances of classes. For example, if we create a butler named \textit{James}, \textit{James} is an instance of the butler class. A class describes a range of behaviors and states an object may have. For example, robot butlers are not good for assembling cars or diving under water because they do not have the correct sensors and manipulators or because they will short out or rust under water.

Pencil is another example of a class. The pencil class is a class of objects that is used for writing. We must be careful to distinguish between class and object. The pencil we have in our hand is a particular instance of the pencil class. The phrase “the pencil in my hand is dull” refers to a particular pencil object. The phrase “bring a pencil to lab” refers to a class of pencil objects.

The behavior and possible states of an object are specified by the \textit{class} to which it belongs. Behavior is defined by the operations a class provides. A class defines the type or kind of an object.

Objects are easy to identify in the physical world because we can see them. They also make their presence known in day-to-day speech as nouns, e.g., James. Operations are usually verbs: move forward, turn, pick up the laundry, etc. Nouns and verbs also give us clues when we look for software objects and their operations.

Software objects are not physical. They cannot be seen, touched, held, or manipulated physically. They are ideas, scratches we make on sheets of paper, the flow of electricity in a computer, or the alignment of magnetic particles on a sheet of plastic. They are conceptual objects that we invent and manipulate in the imaginary world of the computer. Objects may represent boxes on the computer screen, a mailing address, a simulated city, or a way of organizing computer programs in RAM. Similar to \textit{James}, these software objects have an identity, a state, and operations. Software objects can often be recognized because they are nouns, and the operations on these objects are often recognized as verbs.

For example, in an airline reservation system a software object may be a particular flight. It may include operations that allow a travel agent to determine the availability of the flight and add names to and remove names from the reservation list. It may include additional operations that allow an airline to assign a particular plane and crew to the flight. Thus, the state of a flight may include the passenger list, departure and arrival times, destination, airplane, and so on.

Financial planning software may include a stock object. Its state may include the name and number of shares, the current value per share, and the return over the last six months and ten years. Operations may include a method to update a share’s current value, to compute expected earnings, and to assign a desirability index to the stock.
1 Objects

A university registrar’s office may have software to maintain student records. There may be objects that store a record for each student. A student record’s state may include a student number, name, and a list of courses with associated grades. There may be operations to add a course record, change a grade, and compute grade point averages.

1.2.2 Project Requirements

Suppose we are in charge of creating a room-cleaning simulation to help engineers test their robot butler designs and train butler programmers. We are given the following requirements for this project:

Figure 1.1 Initial State of a Room with One Pile of Clothes
A robot butler will start at grid location (11, 3) facing the left (Figure 1.1). It must move forward until it reaches a pile of clothes at grid (8, 3), pick them up, turn around, and leave the room (reach horizontal grid location 11).

Figure 1.1 is the layout of our room. To make things easier for our robot butler, we have laid down a grid on the floor with green tape. We will use this grid to direct how far our butler will move. The butler has special sensors that will allow it to move along grid lines and count how many lines it crosses.

1.2.3 Project Design

An important aspect of software development is the ability to partition a project into objects. We can partition the room-cleaning simulation into three objects (Figure 1.2). The first is the room-cleaning simulation program. A room-cleaning simulation object will be created by a web browser. As a result, we do not have to think about it.

Figure 1.2 Room-Cleaning Simulation Objects

Once started, a room-cleaning simulation object creates two objects of its own: a room and butler. The room object is responsible for determining the size of the simulated window and contains objects that define the locations of the room’s walls and furnishings. The only operations that can be applied to a room are to create a room and start the room-cleaning simulation.

A butler object is more interesting. Butler objects must move, turn, and handle laundry (Figure 1.3). A butler’s state includes its location (myX, myY), the direction it is facing (myDeltaX, myDeltaY), and what it is carrying (myLoad). The Unified Modeling Language (UML) is a graphical notation for describing object-oriented designs. In UML, a class is illustrated with a rectangle partitioned into three components. The top partition contains the name of the class, the middle partition lists the items of state for instances of
1 Objects

Figure 1.3 The Butler class

<table>
<thead>
<tr>
<th>Butler</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>myX</td>
<td>state</td>
</tr>
<tr>
<td>myY</td>
<td>operations</td>
</tr>
<tr>
<td>myDeltaX</td>
<td></td>
</tr>
<tr>
<td>myDeltaY</td>
<td></td>
</tr>
<tr>
<td>myLoad</td>
<td></td>
</tr>
<tr>
<td>forward()</td>
<td></td>
</tr>
<tr>
<td>backward()</td>
<td></td>
</tr>
<tr>
<td>turnRight()</td>
<td></td>
</tr>
<tr>
<td>turnLeft()</td>
<td></td>
</tr>
<tr>
<td>pickUp()</td>
<td></td>
</tr>
<tr>
<td>putDown()</td>
<td></td>
</tr>
<tr>
<td>putIn()</td>
<td></td>
</tr>
<tr>
<td>takeOut()</td>
<td></td>
</tr>
</tbody>
</table>

this class, and the bottom partition lists the operations that can be applied to the class’s objects.

The room-cleaning simulation contains many other objects, but they are hidden from us. For example, Room objects contain walls, beds, desks, baskets, and piles of clothes.

1.2.4 Program Implementation

Our goal is to command our butler to fetch the pile of dirty clothes at grid (8,3) and leave the room. To accomplish this task our butler must move two squares forward, pick up the clothes, make two right-angle turns, and move forward through the door. The program in Listing 1.1 accomplishes this task.

Note: The line numbers in Listing 1.1 are not part of the Java program. They are used throughout the text to make easy reference to the code. Including them within a program will make it fail.

On first viewing, Listing 1.1 appears complex. It is not yet possible to explain the function of every word and character in the program; however, their functions will become clear as we learn more about Java. We will begin by saying that after a web browser creates a RoomCleaningSimulation object it applies the init (initialize) operation to it.
Thus, program execution begins in the init method (Line 15). More details are forthcoming in Chapter 3.

In this chapter we will focus on creating and manipulating objects. We will begin our investigation of Java by focusing on the code that manipulates the room and butler objects. Line 17 creates a room called edsRoom. The code on the right side of the equal sign creates an object of the class Room. The string "onepile.room" is the name of a file that describes the makeup of the room. Once we have created the object we assign it the name edsRoom, which appears on the left side of the equal sign. Thus, the name edsRoom identifies a specific room object.

Once the room is created, we create our butler, named james, in the next line. It is placed at grid location (11,3) in edsRoom. Recall that objects belong to a class and have three properties: identity, operations, and state. The identity of the butler object in Line 18 is

---

Listing 1.1  Pick Up One Pile of Clothes

```java
//**********************************************************
// title:   RoomCleaningSimulation
// author:  © Ed C. Epp - all rights reserved
// date:    4-20-99
//**********************************************************
import java.applet.Applet;
public class RoomCleaningSimulation extends Applet {
   public void init()
   {
      Room   edsRoom = new Room("onepile.room", this);
      Butler james   = new Butler (11, 3, edsRoom);
      edsRoom.waitForStart();
      james.forward();
      james.forward();
      james.pickUp();
      james.turnRight();
      james.turnRight();
      james.forward();
      james.forward();
   }
}
```
1 Objects

james. It belongs to the Butler class. As we mentioned in the previous section, the state of the james object is includes its location, direction, and what it is carrying.

In Line 19 we wait for the start button to be pressed, shown in the upper middle of the display. Once the start button is pressed, the subsequent lines instruct james on how to pick up the dirty clothes.

The operations on a butler are demonstrated in the subsequent lines. First we move forward twice, Lines 20 and 21, by applying the forward operation to james (Figure 1.4). Then our butler picks up our clothes with the pickUp operation (Figure 1.5 A). Next he rotates to the right 90 degrees (Figure 1.5 B). This is followed by another 90 degree rotation (Figure 1.5 C). Finally, our butler moves through the door by moving forward twice.

**Figure 1.4** Move Forward Twice

![Figure 1.4](image1)

**Figure 1.5** A) Pick Up Clothes, B) Turn Right, and C) Turn Right Again

![Figure 1.5](image2)
If a programmer attempts to do an illegal operation, such as instruct the butler to walk through a wall, the simulation stops and displays an error message. Possible errors include walking into another object, trying to put an object down when the butler is not carrying anything, trying to pick up a bed, etc.

This program is a testimony to abstraction. (Abstraction is defined in Chapter 6.) Without abstraction, a butler program would be well beyond our reach. The lines of code within the applet `init` method are made simple because of abstraction. Hidden beneath them are roughly 1,000 lines of Java code. All the complexities, including computing the butler's location and drawing the room, are hidden. But the abstraction does not stop there. The Java design team has hidden many tens of thousands of lines of code within the Java class library that was used to create the butler and room classes. These, in turn, are written on top of other abstractions which are written on top of others. Without these layers of abstraction, the application that we have created would not be possible for a beginning programmer. In the years to come we will begin to peel back these layers of abstraction and learn what each does: from the bits to the operating system, to the language compiler, to the final program.

### 1.2.5 Summary of Operations

Table 1.1 provides us with a summary of some the operations that our butler can perform. There are a few key ideas to keep in mind as we program our butler:

- **Our butler cannot move into any location that contains an object.** If it attempts to, it shuts down.
- **Our butler can carry only clothes and baskets.**
- **Our butler can carry only one thing at a time.** If it is carrying something, it must put the object down before it can carry anything else.
- **A basket can hold many things.** Thus, if our butler is carrying a basket, the basket can contain many piles of clothes or other baskets.
- **The last item inserted into a basket is the first item removed.**
- **A basket is the only thing in which a butler can place objects** (`putIn` and `takeOut` operations).
- **A butler can place objects only on empty grid locations** (`putDown` operation). It cannot place an object on a wall, bed, basket, or desk.
## 1 Objects

<table>
<thead>
<tr>
<th>operation</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>movement</strong></td>
<td></td>
</tr>
<tr>
<td>void forward ()</td>
<td>move forward one square</td>
</tr>
<tr>
<td></td>
<td>error—if motion is blocked by any object</td>
</tr>
<tr>
<td>void forward (int steps)</td>
<td>move forward “steps” squares</td>
</tr>
<tr>
<td></td>
<td>error—if motion is blocked by any object</td>
</tr>
<tr>
<td>void backward ()</td>
<td>move backward one square</td>
</tr>
<tr>
<td></td>
<td>error—if motion is blocked by any object</td>
</tr>
<tr>
<td>void backward (int steps)</td>
<td>move backward “steps” squares</td>
</tr>
<tr>
<td></td>
<td>error—if motion is blocked by any object</td>
</tr>
<tr>
<td>void turnRight ()</td>
<td>turn right 90 degrees</td>
</tr>
<tr>
<td></td>
<td>error—if this butler is carrying something and the location to the right is occupied by anything except a basket</td>
</tr>
<tr>
<td>void turnRight (int count)</td>
<td>turn right 90 degrees &quot;count&quot; times</td>
</tr>
<tr>
<td></td>
<td>error—repeats the error checks above for each 90-degree turn</td>
</tr>
<tr>
<td>void turnLeft ()</td>
<td>turn left 90 degrees</td>
</tr>
<tr>
<td></td>
<td>error—similar to turnRight</td>
</tr>
<tr>
<td>void turnLeft (int count)</td>
<td>turn left 90 degrees &quot;count&quot; times</td>
</tr>
<tr>
<td></td>
<td>error—similar to turnRight</td>
</tr>
<tr>
<td><strong>manipulation</strong></td>
<td></td>
</tr>
<tr>
<td>void pickUp ()</td>
<td>pick up an item</td>
</tr>
<tr>
<td></td>
<td>error—if there are no clothes or basket in front of this butler to pick up or if this butler is already holding something</td>
</tr>
<tr>
<td>void putDown ()</td>
<td>put down the item this butler is holding</td>
</tr>
<tr>
<td></td>
<td>error—if this butler is not holding anything</td>
</tr>
<tr>
<td>void putIn ()</td>
<td>put the item this butler is holding in the basket in front of it</td>
</tr>
<tr>
<td></td>
<td>error—if this butler is not holding anything or there is no basket in front of this butler</td>
</tr>
<tr>
<td>void takeOut ()</td>
<td>take out the item in the basket</td>
</tr>
<tr>
<td></td>
<td>error—if there is no basket in front of this butler, there is no item in the basket, or this butler is already holding something</td>
</tr>
</tbody>
</table>
1.3 Sequential Pattern

Lines 17 through 26 in Listing 1.1 use a fundamental pattern for constructing a program. They show a sequential ordering which specifies the order for executing statements. Christopher Alexander, a prominent American architect, coined the term pattern as a means to communicate common practices for solving building design problems. Software architecture, like building architecture, also contains common ways of doing things. Learning these patterns is of utmost importance for budding software engineers. Patterns communicate ideas that are applied repeatedly to solve problems. Patterns cross computer language boundaries and will serve us for a lifetime.

A pattern describes a commonly used structure for solving a general problem. It contains four parts: a name, a problem, a solution, and a consequence.

Patterns can be large ideas that describe how several objects interact to accomplish a given task, or they can be small ideas that describe how several language statements can be organized to accomplish some task. This text will focus primarily on simple patterns—patterns that describe how a few statements can be organized.

We use patterns in our everyday lives to solve problems. We place convenience stores near residential districts to facilitate quick late-night access. We place supermarkets and hospitals on main thoroughfares to facilitate infrequent but convenient access. We place at arms reach those items we use at our desk most frequently. We place rest rooms near bedrooms to facilitate quick late-night trips. The food and dishes we use most often are placed on lower cupboard shelves. All these strategies are related by a pattern that states that those items needing quick or frequent access are located conveniently.

During the summer we run or do yard work in the morning. We paint the side of the house that is shaded. We water the lawn late in the evening and open our windows at night to let in the cool air. These strategies are related by a pattern of doing during the cool parts of the day those things that are adversely affected by heat.

1.3.1 Name

Each pattern has a name that must concisely communicate the nature of the pattern. The name of the pattern described in this section is “Sequential Pattern.”
1.3.2 Problem

The value of each pattern is described by its ability to solve a problem that occurs repeatedly. For example, the sequential pattern is designed to assure that tasks are accomplished in the correct order. This problem of sequence appears in virtually every computer program.

Most computers are designed so that operations are evaluated sequentially. Each operation is followed by an operation that is executed after the previous one has finished. Most programming languages, as in Java, execute instructions sequentially by default. The order of statements in a program determines the order of execution.

1.3.3 Solution

The solution section of a pattern describes how to solve a recurring problem. We solve the problem of specifying the order in which statements are to be executed by organizing the statements in order. For example, an object must be created before an operation can be applied to it. Thus, we create the object in a statement that precedes using it. If a statement requires a specific situation to hold before it is executed, previous statements must establish that situation.

In the example below, taken from Listing 1.1, statement 1 must be executed before statement 2 because statement 2 relies on the room being created. Statement 3 follows statement 2 because we cannot clean up the room until the room and robot have been created.

```java
1     Room   edsRoom = new Room("onepile.room", this);
2     Butler james  = new Butler (11, 3, edsRoom);
3     edsRoom.waitForStart();
4     james.forward();
5     james.forward();
6     james.pickUp();
7     james.turnRight();
8     james.turnRight();
9     james.forward();
10    james.forward();
```

The order of Lines 4 through 10 is important because they specify the order in which jame accomplishes its task. The butler must be moved in Lines 4 and 5 before it can pick the clothes up in Line 6.

This model assumes that a previous statement has been completely executed before the next statement begins to execute. Execution progresses like a needle and thread being pulled from one statement through another. We can trace that thread back through time.
and unfold a history of statement execution. We say that the thread of execution proceeds from one statement to the next statement.

A flow of control defines a sequence of statements. Each statement, except the first, is preceded by the execution of a single statement. A Java program may have more than one flow of control occurring at the same time. Each is called a thread of execution.

1.3.4 Consequences

Finally, we list the consequences of a pattern. A consequence of the sequential pattern is that we must describe “how” we want something accomplished. We must be aware of the order in which each statement is executed, and we must communicate explicitly step by step, never assuming the computer knows our intent.

Programming the step-by-step process of each object in a program is time consuming. If one step is incorrect, the entire program fails. This style of programming is different from that used in Sojourner. Scientists programmed Sojourner by telling it “what” they wanted it to do; for example, “move to the rock named ‘Yogi’ and take a sample.” Sojourner determined “how” to do it.
Thus far we have examined the fundamental ideas of objects, classes, and operations. The room-cleaning simulation demonstrated how Java may be used to capture these ideas. In this section we will look more closely at Java’s syntax and semantics for expressing objects and operations.

Listing 1.1 has more detail than our description of Java in the following sections will explain. For example, in this chapter we will not explain what an applet is or how it operates. We will not explain the meanings of the public, static, and void keywords. These discussions will follow in the next several chapters.

### 1.4 Syntax and Semantics

In any spoken language there are rules about the construction of utterances. For example, there are rules about which verb forms go with plural nouns. The sentence “The Space Gizzbies are our intergalactic friends.” is syntactically correct. On the other hand, “The Space Gizzbies is our intergalactic friends.” is syntactically incorrect, though people usually understand what we mean even when our grammar is incorrect.

Computer languages also have rules for the construction of programs. But unlike spoken utterances, computer programs must be written in the correct form before a computer can decipher them. The Java compiler will generate syntax errors for programs that are not syntactically correct.

Syntax describes the form that a programming language statement must take. Semantics describes what that statement means.

Semantics can be a problem in spoken and computer languages. For example, in the sentence “The girl takes this with her.” it is not clear to what “this” refers and who the girl is. The statement is syntactically correct but we do not know what it means. Computer programs must not only be correctly stated, they must be semantically clear. A Java compiler cannot initiate the appropriate action by inferring a programmer’s intent. Each program action must be placed in an appropriate context, stated correctly, and meticulously spelled out.

Programs with semantic errors will often be accepted by a Java compiler. However, when run, they will probably produce unexpected results. For example, a program may compute the incorrect insurance premium, cause airplanes to fly upside down, or cause databases to
lose critical information. Even though a program compiles and executes, it is not necessarily correct.

The next several sections will describe some of Java’s syntax and semantics. We will start by looking at some basic syntax building blocks and then move on to elementary statements.

1.4.1 Tokens

The smallest unit of a program that has meaning is a token.

A *token* is a sequence of characters that have a collective meaning (Aho et al).

For example, the following Java statement

\[
\text{Location kansas = redShoes.tap(3);}\\
\]

can be partitioned into the following list of tokens.

\[
\begin{align*}
\text{Location} & \\
\text{kansas} & \\
= & \\
\text{redShoes} & \\
, & \\
\text{tap} & \\
( & \\
3 & ) \\
; & \\
\end{align*}
\]

Spaces, tabs, and new lines, which collectively are called *white space*, act as token delimiters. *Delimiters* separate tokens. Without the space between Location and kansas the Java compiler would interpret Locationkansas as one token. Some tokens do not need white space to delimit them. For example, kansas=redShoes would be interpreted as three tokens. The rules for token delimitation may not be clear at first. For example, hike123 would be interpreted as one token while 123hike would represent two tokens. The rules will become clear with time. If you are not sure, extra white space between tokens has no impact. For example, the following line is equivalent to the one above.

\[
\begin{align*}
\text{Location kansas} & = \text{redShoes .tap(} \\
2 & ) \\
; & \\
\end{align*}
\]
Careful and consistent use of white space makes a program easier to understand. Observe how various lines in example programs are indented. Indention provides clues about how a program is organized, and also makes a program attractive. We will describe the rules used as we move through this text. For now, follow the form of the example code or the rules provided by your instructor.

Spaces within a token are usually not permitted. For example, in the following, red and Shoes are interpreted as two tokens when the intent is to have only one.

```java
Location kansas = red Shoes.tap(3); // error
```

Each token has an important semantic meaning that, when combined with other tokens, performs a useful action. One may guess from the previous Java statement that its intent is to take us back to Kansas by tapping three times with our red shoes. “Kansas” and “red shoes” identify key objects of our action. Tap identifies an operation.

### 1.4.2 Identifiers

Identifiers, such as `redShoes`, `kansas`, `forward`, `tap`, and `RoomCleaningSimulation`, are arbitrary. They could be replaced everywhere with `v`, `w`, `x`, `y`, and `z` with no ill effects, except that it would make the intent of a program more difficult to understand. Identifiers should make a program’s intent easy to infer.

An **identifier** is a name given to an object, class, or method (operation). Identifiers may contain upper- and lowercase letters, the digits 0 through 9, the underscore character (`_`), and the dollar sign (`$`). An identifier may not begin with a digit.

Valid identifiers include `R2D2` and `Plea$eSendMoney`. Invalid identifiers include `3PO` and `%Done`. Java reserves some keywords, for example `public`, `class`, and `extends`, that cannot be used as identifiers (see Section 1.4.3).

The naming convention used in this book is that class names begin with an uppercase letter. Object and method names begin with a lowercase letter. Instance variables that define the state of an object begin with the prefix “my.” These conventions are not enforced by the compiler but rather chosen by the author to make programs easier to read. Your instructor or employer may have a different coding standard that you are expected to follow.
Java is case sensitive. It makes a difference how identifiers and keywords are capitalized. For example, it is necessary that `import` is in all lowercase letters—the Java compiler will generate a syntax error if any letter in `import` is capitalized. Some capitalization errors will allow the program to compile but it will not execute correctly. For example, replacing the following line with Line 15 in Listing 1.1 will result in a valid program.

```java
public void Init ()      // program will not run
```

Typing `Init` instead of `init` will not generate a syntax error. However, the program will not do anything when executed. A capitalization error can be frustrating because the program looks correct but doesn’t perform.

### 1.4.3 Keywords

*Keywords* are special words that look like identifiers. Java gives special meaning to each keyword. Attempting to use it as an identifier will confuse the compiler; do not attempt to use keywords in this way. The list of Java keywords is shown in Table 1.2.

**Table 1.2** Java Keywords

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>do</td>
<td>implements</td>
<td>package</td>
</tr>
<tr>
<td>boolean</td>
<td>double</td>
<td>import</td>
<td>private</td>
</tr>
<tr>
<td>break</td>
<td>else</td>
<td>inner</td>
<td>protected</td>
</tr>
<tr>
<td>byte</td>
<td>extends</td>
<td>instanceof</td>
<td>public</td>
</tr>
<tr>
<td>case</td>
<td>final</td>
<td>int</td>
<td>rest</td>
</tr>
<tr>
<td>cast</td>
<td>finally</td>
<td>interface</td>
<td>return</td>
</tr>
<tr>
<td>catch</td>
<td>float</td>
<td>long</td>
<td>short</td>
</tr>
<tr>
<td>char</td>
<td>for</td>
<td>native</td>
<td>static</td>
</tr>
<tr>
<td>class</td>
<td>future</td>
<td>new</td>
<td>super</td>
</tr>
<tr>
<td>const</td>
<td>generic</td>
<td>null</td>
<td>switch</td>
</tr>
<tr>
<td>continue</td>
<td>goto</td>
<td>operator</td>
<td>synchronized</td>
</tr>
<tr>
<td>default</td>
<td>if</td>
<td>outer</td>
<td>this</td>
</tr>
</tbody>
</table>
1 Objects

1.5 Creating Objects

1.5.1 Declaration Statements with Constructors

Tokens are combined into statements that express a complete action. Statements are typically terminated with a semicolon token.

Instructions are communicated to the computer through statements. A statement is a sequence of tokens that collectively express an action.

The first statement we will investigate is a declaration statement. We must be able to create and name objects. The form of a statement that will accomplish this is shown below.

\[
\text{ClassIdentifier ObjectIdentifier} = \text{new Constructor} ;
\]

For example:

\[
\begin{align*}
\text{Butler} & \quad \text{james} = \text{new Butler (11, 3, edsRoom)}; \\
\text{LunarLander} & \quad \text{eagle} = \text{new LunarLander (tranquility)}; \\
\text{Shoes} & \quad \text{redShoes} = \text{new Shoes (Color.red)}; \\
\text{Rectangle} & \quad \text{box} = \text{new Rectangle (100, 200, 13, 39)};
\end{align*}
\]

The class identifier indicates the class to which an object identifier may refer. For example, the second statement above specifies that the object identifier \text{eagle} may refer only to objects that belong the class \text{LunarLander}.

The part of the example code in the above statements after the equal token creates a new object. The first statement creates a butler named \text{james} at grid location (11,3) in edsRoom. The second statement creates a lunar lander on the moon at the Sea of Tranquility. The third statement creates a red shoe. The fourth statement creates a rectangle which is 13 units wide, 39 units high, at location (100,200).

An object identifier may be declared only once. For example, in the following code fragment, an attempt is made to create two objects with the name \text{james}. The second attempt will generate an error message.

\[
\begin{align*}
\text{Butler} & \quad \text{james} = \text{new Butler (11, 3, edsRoom)}; \quad \text{// ok} \\
\text{Butler} & \quad \text{james} = \text{new Butler (4, 7, edsRoom)}; \quad \text{// not ok} \\
\text{Butler} & \quad \text{franz} = \text{new Butler (8, 2, edsRoom)}; \quad \text{// ok}
\end{align*}
\]

The part of the above statements after the \text{new} keyword is called the constructor. For example, a constructor for a butler looks like \text{Butler (11, 3, edsRoom)}. 
Creating Objects

A constructor is a special operation (method) that creates an object when the `new` operator is applied to it. This process is called instantiating an object.

The `new` operator is an example of an operation that is applied to a class rather than an object. It invokes a constructor method that describes how the object is to be created. Constructors always have the same name as the name of the class they are creating.

Each constructor may have zero or more parameters. A parameter is a mechanism for sending special information to the constructor that customizes the creation of an object. For example, the parameter `tranquility` in the `LunarLander` constructor specifies that the lander is to be created on the moon at the Sea of Tranquility. The `Color.red` parameter in the `Shoes` constructor specifies that a red shoe is to be created. Finally, the `39` in the `Rectangle` constructor specifies that a 39-unit-wide face is to be created.

A parameter is a mechanism for passing data (information) from the current method to another method.

Finally, the equal sign specifies that the identifier on its left side is to refer to the object on the right side of the equal sign. The equal sign represents assignment, not equality. For example, the lunar lander created by the constructor `LunarLander (tranquility-Base)` is assigned to the identifier `eagle`. (See Figure 1.6 for a summary of the parts of a declaration statement.)

![Figure 1.6 Declaring and Creating Objects](image)

1.5.2 Declaration Statement without a Constructor

The declarations described in the previous section can be partitioned into two statements: a declaration without a constructor followed by an assignment statement. For example:
Butler james;
   james = new Butler (11, 3, edsRoom);

The form of declarations without constructors is shown below.

   ClassIdentifier  ObjectIdentifier  ;

For example:

   Butler       james;
   LunarLander  falcon;
   Shoes        mudShoes;
   Rectangle    wall;

Because only names are introduced and there are no constructors, no objects are created. For example, the above states that the identifier *falcon* can be used to refer to a lunar lander, but that it currently refers to no object. When an object identifier does not refer to an object, we say that its value is *null*.

A declaration statement merely declares that a particular identifier can be associated with specific classes of objects. A common error is to declare an object identifier without associating an object with it. Applying an operation to such a identifier will result in an error. A good rule of thumb is to assign an object to an identifier whenever one is declared.

### 1.5.3 Assignment Statement

An *assignment* statement is used to give an object an identity.

   identifier   =  object   ;

For example:

   darkWing = new SpaceCruiser();
   falcon   = darkWing;

The above statements create a single object that belongs to the space cruiser class. This single object has two identities. It can be referred to as *darkWing* or *falcon*. The clue that there is only one object is that the new operator is applied to a constructor one time. Thus, a single object may have more than one identifier that refers to it. This is called *aliasing*. Thus, *falcon* and *darkWing* are aliases for the same object.

Giving a single object more than one name is called *aliasing*.

An object identifier can be assigned objects many times. However, an identifier can reference only one object at a time. Only the last assignment is retained. For example:
Applying Operations to Objects

LunarLander eagle = new LunarLander (tranquility);
eagle = new LunarLander (storms);
LunarLander falcon = new LunarLander (descartes);
eagle = falcon;
eagle = new LunarLander (apennine);

Four objects are created. Each is assigned to the identifier eagle. Only the last reference is retained. The identifier, eagle, refers only to the lander at Apennine. The first two objects are lost to us since we have no way of referring to them. Eventually a garbage collector will come along and recycle them.

Objects require space resources in RAM (random-access memory—computer memory). The more objects we create, the more space is used. A garbage collector is an action taken automatically by a program to return the resources required to maintain an object. It allows these resources to be reused. A garbage collector is typically invoked when resources become low.

1.6 Applying Operations to Objects

1.6.1 Invoking Methods

Once an object is created, we must apply operations to it. Recall that objects have three important properties: identity, state, and operations. Methods are the mechanisms for doing operations. A method is often used to access or modify an object’s state. One form for invoking a method on an object is as follows:

ObjectIdentifier.MethodIdentifier ( parameter1, parameter2, ... ) ;

The following examples show how a method can be used to change the state of an object.

james.forward();
james.turnRight();
franz.forward(3);

In the first example, the method forward requests that our butler james move forward (Figure 1.7.) This changes the state of the object james by changing its location. In the next example, the butler named james is requested to turn 90 degrees to the right. In the final example, the butler named franz moves forward three grid locations. Additional operations that can be applied to butlers are shown in Table 1.1 on page 12.

The classic means for expressing the first example above is to say, “the message forward is sent to the james object.” Another way of saying this is that the james object is commanded to move forward.
1 Objects

A *method* is an operation. Methods may change or access the state of an object.

**Figure 1.7 Applying the forward Operation to james**

1.6.2 A Common Null Pointer Error

A common error is to try to manipulate an object that has not been created. For example:

```java
Butler james;
james. forward();    // error - null pointer exception
```

In the example above `james` is supposed to reference an object, but it does not. The identifier, `james`, is said to be *null*. An error occurs when we try to apply the `forward` operation to an object called `james`. No object was created, so there is no object on which to apply an operation, and the Java program will fail with a null pointer exception.

An identifier that does not refer to an object is said to be *null*. A null pointer exception occurs when we try to manipulate an object that does not exist.

The problem is fixed by creating an object.

```java
Butler james  = new Butler(11, 3, edsRoom);
james.forward();
```

1.7 Comments

Comments are meant for human consumption and are ignored by a Java compiler. They may be used to indicate who wrote a program, why it was written, and how it functions. Even though comments are not necessary for a program to function, their importance cannot be overemphasized. A program will go through many revisions. Comments furnish critical information to other programmers who must maintain a program.

A comment begins with a double forward slash (`//`) and extends to the end of the line.
Comments are one tool in a strategy to make programs easier to read. Several guidelines for making a program easier to read include:

- Use comments and white space to partition code into blocks of code that accomplish some purpose.

```java
// move james to the first pile of laundry
james.forward();
james.forward();
james.turnRight();
james.forward();

// james puts the laundry into the basket
james.pickUp();
james.turnRight();
james.forward();
james.forward();
james.forward();
james.putIn();
```

- Comments should give insight that looking at the code does not reveal. Comments that repeat what the code says are not useful.

```java
// a comment that helps explain the code
x = a * b;   // find the area of the display window

// an example of a distracting comment
x = a * b;   // x gets a multiplied by b
```

- Try to write code that does not need comments to understand. Using the previous example, we can make the code easy to understand by using identifier names that help make the code self-documenting. For example, the identifier names below help communicate the code’s intent:

```java
displayWindowArea = width * height;
```

## 1.8 Compiling and Running Applets

An *applet* is a Java program that can be run from a Java-aware web browser or applet viewer. We will look at applets more carefully in Chapter 3. In this section we will learn how to set up an applet so that we can execute it.

### 1.8.1 Compiling an Applet

The complete source code for a robot program is in Listing 1.1. When this program is executed, the window in Figure 1.1 is displayed. The simulation is started with a press of the “Start” button.
1 Objects

Listing 1.1 is stored in a file named RoomCleaningSimulation.java. The name of the file must match the name of the class (see Line 13) followed by the java extension. Once the cleaning program is entered in the RoomCleaningSimulation.java file, it is compiled. Compiling checks the Java listing for syntax errors. If no syntax errors are found, a byte-code representation of the program is created. Java byte-codes allow the efficient execution of a Java program on many different kinds of computers.

The method used for compiling a Java applet varies by computer platform and software package. Compilation often takes place with the click of a button. The process of compiling a Java applet will create a new file. In this example, the new file will be called RoomCleaningSimulation.class. (See Figure 1.8.) This file has the platform-independent byte-codes that can be interpreted by a web browser or applet viewer.

A translator is a computer program that converts one computer language to another. A Java compiler is a translator that converts Java programs to byte-code instructions. Java programs are designed to be written, read, and modified by humans. They must conform to the Java syntax exactly before they can be translated into byte-code. Byte-code is designed to be easy to interpret by computers. However, just because a program has correct syntax and is translated into byte-codes does not make the program correct. The Java compiler does not check for correct semantics. That must be done through program testing.

**Figure 1.8** Compiling Java Programs
1.8.2 Including an Applet on a Web Page

Once a program is compiled it can be executed by viewing it through a Java-aware web browser or by using an applet viewer. The browser or applet viewer requires an HTML file with a link to the compiled program. For example, opening the RoomCleaningSimulation.html file in Listing 1.2 with a Java-aware web browser or with an applet viewer will run the lander simulation. This process is illustrated in Figure 1.9.

When a Java-aware web browser accesses a page that references the cleaning simulation class, the browser will first create a lander simulation object and then apply the init operation to it. Most Java software development environments bring up an applet viewer with a click of a button. Applet viewers allow us to execute a Java program without a web browser.

**Figure 1.9** Accessing an Applet with a Web Browser

An *applet viewer* is a stripped-down web browser. Its purpose is for viewing local Java applets.
Byte-code is the key to running Java programs over the network. No matter whether one has a Solaris, a Macintosh, or a Windows computer, byte-code will run on the computer if it has an appropriate byte-code interpreter.

An interpreter is a program that executes a program that has not been converted into the native language of a computer. The native languages of Solaris, Macintosh, and Windows 98 computers are not byte-codes. Since each of these machines has a different native language, a new language (byte-code) was created and implemented on each machine.

The mechanism that allows cross-platform execution is an agreed-upon intermediate language, such as byte-code, that is interpreted on each machine. Interpreted code generally runs about 20 times slower than code that is translated into the native computer’s machine code, but this is the price for cross-platform execution. However, there are byte-code compilers that translate byte-code into native machine code. Of course, a program compiled to machine code on a Sparc will only run on other Sparcs. The same constraints apply to PCs and Macs.

The room cleaning simulation can be run from a Java-aware browser using the web page shown in Listing 1.2 on page 28. The key line is:

```html
<applet code=RoomCleaningSimulation.class width=100 height=50>
</applet>
```

The above tag states that the applet byte-code can be found in the file RoomCleaningSimulation.class. In addition, the web page reserves a space 100 pixels wide and 50 pixels high for the applet.

**Listing 1.2  Room Cleaning Simulation Web page**

```html
1 <title>Room Cleaning</title>
1 <hr>
1 <applet code=RoomCleaningSimulation.class width=100 height=50>
1 </applet>
1 <hr>
1 <a href="RoomCleaningSimulation.java">The source.</a>
```
1.9 Summary

An object has identity, state, and operations. Its identity is a name we use to reference an object. Its state indicates something about an object’s condition, mode, status, or situation. An operation is a command that when applied to an object will change its state or report its state. An object is an instance of a class. Classes specify the behavior of a type of object.

Patterns describe common ways of doing things. They are commonly used structures for solving general problems. Patterns communicate ideas that are applied repeatedly to solve problems. Patterns cross computer language boundaries and will serve us for a lifetime.

When the sequential pattern is used, program statement are executed in the order in which they are listed.

The smallest elements of a computer program are tokens. They are combined to create statements. Syntax describes the form that a computer language statement must take. Semantics indicate what it means.

In Java we create objects using the following syntax:

```
ObjectClass objectIdentifier = new ObjectClass (parameters) ;
```

Operations are applied to an object using the following syntax:

```
objectIdentifier.method(parameters);
```

Applets are Java classes. A Java-aware web browser can use applet classes to create objects that can run within the web browser. Applets have operations. For example, the `init` operation executes when a web browser creates an applet.

1.10 Bibliography

  
  Often referred to as the “Dragon Book,” this was used as a starting point for many of the definitions for syntactic elements.
1 Objects

  A good intermediate introduction to object-oriented methodology. It was used as a starting point for object definitions.

  The Butler class has its roots here.
1.11 Exercises

Objects and Classes—Section 1.2

1 For each of these classes, identify a specific object that belongs to that class, and list operations on it.
   a. CD player
   b. sweatshirt
   c. pencil
   d. mystery novel
   e. garden
   f. hammer
   g. oxygen
   h. French door

2 For each of these objects, name a class to which it belongs.
   a. Empire State Building
   b. Washington Monument
   c. Grand Canyon
   d. Madonna
   e. Spirit of St. Louis
   f. ET
   g. Sol
   h. Oregon

3 It is your job to assist in designing a university registration system. It needs to contain the object classes listed below. For each class, list the items that make up its state and describe available operations.
   a. student record
   b. course record
   c. instructor record
   d. classroom
1 Objects

It is your job to assist in designing a personal finance system. It needs to contain the object classes listed below. For each class, list the items that comprise its state and describe available operations.

a. bank statement
b. car loan
c. saving statement

How would you change the program in Listing 1.1 on page 9 if \textit{james} began at (11,2)?

How would you change the program in Listing 1.1 on page 9 if the pile of clothes were initially located at (8,2)?

Invent an additional operation on the Butler class. Describe what it should do.

Sequential Pattern—Section 1.3

Write down a sequence of steps you commonly perform that must be completed sequentially. For example, the following can be completed only within the following order:

a. Take the cap off the toothpaste tube.
b. Put toothpaste on the toothbrush.
c. Brush teeth.

Syntax—Section 1.4

Which of the following are valid identifiers:

a. fFarPoint
b. farPoint
c. FarPoint
d. far_point
e. _far_point_
f. far-point
g. Boeing747
h. switch
i. 3Corners
j. $$$
k. star*
10 Partition the statements in Exercise 12 into tokens.

11 Identify the tokens, identifiers, and keywords in the statements in Exercise 12. Some items may have more than one designation, for example, token and identifier.

Creating Objects—Section 1.5 / Applying Operations, Section 1.6

12 For each of the following, name the class, the object, the constructor, the name of the operation in the second line, and any parameters.
   a. Moon europa = new Moon();
      europa.shine();
   b. Graphics g;
      g.drawLine(2,3,7,1);
   c. Stack plates = new Stack(100);
      plates.pop(5);
   d. Door front = new Door(oak, 36);
      front.lock();
   e. Account mine = new Account("Elmer Snid", 1845.12);
      mine.deposit(351.49);
   f. w x = new w();
      x.y(z);

13 Making sure your syntax is correct, create Java code fragments that will do the following:
   a. Create a dog and assign it to the identifier "duke". Apply the "roll over" operation to it.
   b. Create a super hero and assign it to the identifier "flashGordon". Apply the "fly" operation to it.
   c. Create a cake for Ema Lou's birthday. Apply the "bake" operation to it with a parameter of 350 degrees.

14 Identify the syntax error in each of the following statements.
   a. CDPlayer my Player = new CDPlayer();
   b. Car theWreck = new Car()
   c. Point p = New Point (40, 50);
   d. Box small = new Box (50 25 50 100);
   e. Color new = new Color (255, 255, 255);
1 Objects

f. Thing mine = new Thing;
g. Robot 3PO = new Robot();
h. piano.play();
i. myCar.move forward (30);
j. computer number2.shutDown();

1.12 Projects

Compiling and Executing Java Applets—Section 1.8

15 Practice using your Java development system by compiling and running the butler program in Listing 1.1.

Robot Control—Section 1.2

16 Program james, our butler, to move through the maze stored in the file chap1a.room (Figure 1.10). Begin james at grid (6,3) and move him to grid (1,0). The beginning of a program to accomplish this task is shown in Listing 1.3.

Figure 1.10 The Maze in the File chap1a.room
Our parents just called from the front desk telling us they are on the way up. We need to hide our laundry behind the bed. Have James move our laundry from (6,1) to (1,4) and then have him leave the room. Change Line 8 in Listing 1.3 to start James at (11,3) and Line 7 to open the room file in chap1b.room (Figure 1.11).

**Figure 1.11** The Room in the File chap1b.room
Our parents are on their way up again. This time we have laundry all over the room. Have James move all of our laundry at (1,3), (5,2), (6,1), (7,2), and (10,7) into the laundry basket that is at (1,1). Next, have James disappear out the door with our basket of dirty clothes. Since we are in a hurry, have James do this in the fewest possible steps. It may expedite James to have him move the laundry basket to several intermediate locations. James must put down the laundry basket before he can pick up clothes and put them in the basket. Start James at (11,3) and use the room in file chap1d.room (Figure 1.12).

Figure 1.12 The Room in the File chap1d.room