01: Assist. Lec. Dhafer T. Shihab

Getting Started with Java

The thing that Java tries to do and is actually remarkably successful at is spanning a lot of different domains, so you can do app server work, you can do cell phone work, you can do scientific programming, you can write software, do interplanetary navigation, all kinds of stuff…

—Java language creator James Gosling

When the Java programming language was unleashed on the public in 1995, it was an inventive toy for the Web that had the potential to be more.

The word "potential" is a compliment that comes with an expiration date. Sooner or later, potential must be realized, or new words and phrases are used in its place, such as "slacker," "letdown," "waste," or "major disappointment to your mother and me."

As you develop your skills in this book's 21 one-day tutorials, you'll be in a good position to judge whether the language has lived up to more than a decade of hype.

You'll also become a Java programmer with a lot of potential.

The Java Language

Now in its ninth major release, Java has lived up to the expectations that accompanied its arrival. More than four million programmers have learned the language and are using it in places such as NASA, IBM, Kaiser Permanente, and Google. It's a standard part of the academic curriculum at many computer science departments around the world. First used to create simple programs on web pages, Java can be found today in the following places (and many more):

- Web servers
- Relational databases
- Orbiting telescopes
- E-book readers
- Cell phones

Although Java remains useful for web developers, its ambitions today extend far beyond the Web. Java has matured into one of the most popular general-purpose programming languages.

History of the Language

The story of the Java language is well known by this point. James Gosling and a team of developers were working on an interactive TV project at Sun Microsystems in the mid-1990s when Gosling became frustrated with the language being used. C++ was an object-oriented programming language developed a decade earlier as an extension of the C language.

To address some of the things that frustrated him about C++, Gosling holed up in his office and created a new language that was suitable for his project.

Although that interactive TV effort flopped, Gosling’s language had unforeseen applicability to a new medium that was becoming popular at the same time: the Web.

Java was released to the public for the first time in 1995. Although most of the language’s features were primitive compared with C++ (and Java today), special Java programs called applets could be run as part of web pages on the most popular web browser at that time, Netscape Navigator.

This functionality—the first interactive programming available on the Web—drew so much attention to the new language that several hundred thousand programmers learned Java in its first six months.

Even after the novelty of Java web programming wore off, the overall benefits of the language became clear, and the programmers stuck around. There are more professional Java programmers today than C++ programmers.

Sun Microsystems controlled the development of the Java language from its inception until 2010, when the company was acquired by the database and enterprise software giant Oracle in a $7.4 billion deal. Oracle, a longtime user of the language on its own products, has a strong commitment to supporting Java and continues to increase its capabilities with each new release.

Introduction to Java

Java is an object-oriented, platform-neutral, secure language designed to be easier to learn than C++ and harder to misuse than C and C++.

Object-oriented programming (OOP) is a software development methodology in which a program is conceptualized as a group of objects that work together. Objects are created from templates called *classes*, and they contain data and the statements required to use that data. Java is primarily object-oriented, as you’ll see later today when you create your first class and use it to create objects.

Platform neutrality is a program’s ability to run without modification in different computing environments. Java programs are transformed into a format called *bytecode* that can be run by any computer or device equipped with a Java Virtual Machine (JVM). You can create a Java program on a Windows 10 machine that runs on a Linux web server, an Apple Mac using OS 10.10, and a Samsung Android phone. As long as a platform has a JVM, it can run the bytecode.

Although the relative ease of learning one language over another is always a point of contention among programmers, Java was designed to be easier than C++ primarily in the following ways:

- Java automatically takes care of memory allocation and deallocation, freeing programmers from this error-prone and complex task.
- Java doesn’t include pointers, a powerful feature for experienced programmers that can be easily misused and introduce major security vulnerabilities.
- Java includes only single inheritance in object-oriented programming.

The lack of pointers and the presence of automatic memory management are two key elements of Java security.

Selecting a Development Tool

Now that you've been introduced to Java as a spectator, it's time to put some of these concepts into play and create your first Java program.

Before you get started, you must have software on your computer that can be used to edit, prepare, and run Java programs that use the most up-to-date version of the language: Java 8.

Several popular integrated development environments (IDEs) for Java support version 8, including IntelliJ IDEA and the open source software Eclipse.

If you are learning to use these tools at the same time as you are learning Java, it can be a daunting task. Most IDEs are aimed primarily at experienced programmers who want to be more productive, not new people who are taking their first foray into a new language.

The simplest tool for Java development is the Java Development Kit, which is free and can be downloaded from www.oracle.com/technetwork/java/javase/downloads.

Whenever Oracle releases a new version of Java, it also makes a free development kit available over the Web to support that version. The current release is Java SE Development Kit 8.(or newer)

The drawback of developing Java programs with the JDK is that it is a set of command-line tools. Therefore, it has no graphical user interface for editing programs, turning them into Java classes, and testing them. (A command line is simply a prompt for typing text commands. It's available in Windows as the program Command Prompt.)

Oracle offers an excellent free IDE for Java programmers called NetBeans on the website www.netbeans.org. Because NetBeans is easier to use for most people than the JDK, it's employed throughout this lectures.

As soon as you have a Java development tool on your computer that supports Java 8, you're ready to dive into the language.

If you don't have one on your computer yet, now's the time to set one up—preferably NetBeans.

Tip

For more information on the other IDEs for Java, visit the IDEA site at www.jetbrains.com/idea and Eclipse at www.eclipse.org.

Object-Oriented Programming

The biggest challenge for a new Java programmer is learning object-oriented programming while learning the Java language.

Although this might sound daunting if you are unfamiliar with this style of programming, think of it as a two-for-one discount for your brain. You will learn object-oriented programming by learning Java. There's no other way to make use of the language.

Object-oriented programming is an approach to building computer programs that mimics how objects are assembled in the physical world.

By using this style of development, you can create programs that are more reusable, reliable, and understandable.

To get to that point, you first must explore how Java embodies the principles of object-oriented programming.

If you already are familiar with object-oriented programming, much of today's material will be a review for you. Even if you skim over the introductory material, you should create the sample program to get some experience in developing, compiling, and running Java programs.

There are many ways to conceptualize a computer program. One way is to think of a program as a series of instructions carried out in sequence, which commonly is called *procedural programming*. Some programmers start by learning a procedural language such as a version of BASIC.

Procedural languages mirror how a computer carries out instructions, so the programs you write are tailored to the computer's manner of doing things. One of the first things a procedural programmer must learn is how to break a problem into a series of simple steps followed in order.

Object-oriented programming looks at a computer program from a different angle, focusing on the task the program was created to perform, not on how a computer handles tasks.

In object-oriented programming, a computer program is conceptualized as a set of objects that work together to accomplish a task. Each object is a separate part of the program, interacting with the other parts in highly controlled ways.

For a real-life example of object-oriented design, consider a stereo system. Most systems are built by hooking together a bunch of different objects, which are more commonly called components. If you came back from a stereo shopping trip, you might bring home all these objects:

- Speaker components that play midrange and high-frequency sounds.
- A subwoofer component that plays low bass frequency sounds.
- A tuner component that receives radio broadcast signals.
- A CD player component that reads audio data from CDs.
- A turntable component that reads audio data from vinyl records.

These components are designed to interact with each other using standard input and output connectors. Even if you bought speakers, subwoofer, tuner, CD player, and turntable made by different companies, you could combine them to form a stereo system—as long as each component has standard connectors.

Object-oriented programming works under the same principle: You put together a program by creating new objects and connecting them to each other and to existing objects provided by Oracle or another software developer. Each object is a component in the larger program, and they are combined together in a standard way. Each object plays a specific role in the larger program.

An *object* is a self-contained element of a computer program that represents a related group of features and that is designed to accomplish specific tasks.

Objects and Classes

Object-oriented programming is modeled on the observation that in the physical world, objects are made up of many kinds of smaller objects.

The capability to combine objects is only one aspect of object-oriented programming. Another important feature is the use of classes.

A *class* is a template used to create an object. Every object created from the same class has similar features.

Classes embody all features of a particular set of objects. When you write a program in an object-oriented language, you don't define individual objects. Instead, you define classes used to create those objects.

If you were writing a networking program in Java, you could create a `HighSpeedModem` class that describes the features of all Internet modems. These devices have the following common features:

- They connect to a computer's ethernet port.
- They send and receive information.
- They communicate with Internet servers.

The `HighSpeedModem` class serves as an abstract model for the concept of such a modem. To have something concrete you can manipulate in a program, you need an object. You must use the `HighSpeedModem` class to create a `HighSpeedModem` object. The process of creating an object from a class is called *instantiation*, which is why objects also are called *instances*.

A `HighSpeedModem` class can be used to create different `HighSpeedModem` objects in a program, each with different features, such as the following:

- Some function as a wireless Internet gateway, whereas others do not.
- Some can be used as a network router.
- They support different connection speeds.

Even with these differences, two `HighSpeedModem` objects still have enough in common to be recognizable as related objects.

Here's another example: Using Java, you could create a class to represent all command buttons—the clickable rectangles that appear on windows, dialogs, and other parts of a program's graphical user interface.

When the `CommandButton` class is developed, it could define these features:

- The text displayed on the button
- The size of the button
- Aspects of its appearance, such as whether it has a 3D shadow

The `CommandButton` class also could define how a button behaves when it is clicked.

After you define the `CommandButton` class, you can create instances of that button—in other words, `CommandButton` objects. The objects all take on the basic features of a button as defined by the class. But each one could have a different appearance and slightly different behavior, depending on what you need that object to do.

By creating a `CommandButton` class, you don't have to keep rewriting the code for each button you want to use in your programs. In addition, you can reuse the `CommandButton` class to create different kinds of buttons as you need them, both in this program and in others.

When you write a Java program, you design and construct a set of classes. When your program runs, objects are created from those classes and used as needed. Your task as a Java programmer is to create the right set of classes to accomplish what your program needs to accomplish.

Fortunately, you don't have to start from scratch. The Java language includes the Java Class Library, more than 4,000 classes that implement most of the functionality you will need. These classes are installed along with a development tool such as the JDK.

When you're talking about programming in the Java language, you're actually talking about using this class library and some standard keywords and operators defined in Java.

The class library handles numerous tasks, such as mathematical functions, text, graphics, user interaction, and networking. Working with these classes is no different from working with the Java classes you create.

For complicated Java programs, you might create a whole set of new classes that form their own class library for use in other programs.

Reuse is one of the fundamental benefits of object-oriented programming.

Note

In the Java Class Library, one of Java's standard classes, `JButton` in the `javax.swing` package, encompasses all the functionality of this hypothetical `CommandButton` example, along with a lot more.

Attributes and Behavior

A Java class consists of two distinct types of information: attributes and behavior.

Both of these are present in `MarsRobot`, a project you will implement today as a class. This project, a simple simulation of a planetary exploration vehicle, is inspired by the Mars Exploration Rovers used by NASA's Jet Propulsion Laboratory program to do research on the surface and geology of the planet Mars.

Before you create the program, you need to learn some things about how object-oriented programs are designed in Java. The concepts may be difficult to understand as you're introduced to them, but you'll get plenty of practice with them throughout the lectures.

Attributes of a Class of Objects

Attributes are the data that differentiate one object from another. They can be used to determine the appearance, state, and other qualities of objects that belong to that class.

An exploration vehicle could have the following attributes:

- **Status**—Exploring, moving, returning home
- **Speed**—Measured in miles per hour
- **Temperature**—Measured in degrees Fahrenheit

In a class, attributes are defined by *variables*—places to store information in a computer program. *Instance variables* are attributes that have values that differ from one object to another.

An instance variable defines an attribute of one particular object. The object's class defines what kind of attribute it is, and each instance stores its own value for that attribute. Instance variables also are called *object variables* or *member variables*.

Each class attribute has a single corresponding variable. You change that attribute of the object by changing the value of the variable.

For example, the `MarsRobot` class defines a `speed` instance variable. This must be an instance variable because each robot travels at a different speed. The value of a robot's `speed` instance variable could be changed to make the robot move more quickly or slowly.

Instance variables can be given a value when an object is created and then stay constant throughout the life of the object. They also can be given different values as the object is used in a running program.

For other variables, it makes more sense to have one value that is shared by all objects of that class. These attributes are called *class variables*.

A class variable defines an attribute of an entire class. The variable applies to the class itself and to all its instances, so only one value is stored, no matter how many objects of that class have been created.

An example of a class variable for the `MarsRobot` class would be a `topSpeed` variable that holds the maximum speed any robot is capable of traveling. If an instance variable were created to hold the speed, each object could have a different value for this variable. That could cause problems because no robot is capable of exceeding it.

Using a class variable prevents this problem because all objects of that class share the same value automatically. Each `MarsRobot` object would have access to that variable.

Behavior of a Class of Objects

Behavior refers to the things that a class of objects can do—both to themselves and to other objects. Behavior can be used to change an object's attributes, receive information from other objects, and send messages to other objects, asking them to perform tasks.

A Mars robot could have the following behavior:

- Check the current temperature
- Begin a survey
- Accelerate or decelerate its speed
- Report its current location

Behavior for a class of objects is implemented using methods.

Methods are groups of related statements in a class that perform a specific task. They are used to accomplish specific tasks on their own objects and on other objects and are comparable to functions and subroutines in other programming languages. A well-designed method performs only one task.

Objects communicate with each other using methods. A class or object can call methods in another class or object for many reasons, including the following:

- To report a change to another object
- To tell the other object to change something about itself
- To ask another object to do something

For example, two Mars robots could use methods to report their locations to each other and avoid collisions, and one robot could tell another to stop so that it can pass by safely.

Just as there are instance and class variables, there also are instance and class methods. Instance methods, which are usually just called methods, are used when you are working with an object of the class. If a method changes an individual object, it must be an instance method. Class methods apply to a class itself.

Creating a Class

To see classes, objects, attributes, and behavior in action, you will develop a `MarsRobot` class, create objects from that class, and work with them in a running program.

Note

The main purpose of this project is to explore object-oriented programming.

This book uses NetBeans as its primary development tool for creating Java programs. NetBeans organizes Java classes into projects. It will be useful to have a project to hold the classes you create in this book. If you have not done so already, create a project:

1. Choose the menu command File, New Project. The New Project dialog appears.
2. In the Categories pane, choose Java.
3. In the Projects pane, choose Java Application and click Next. The New Java Application dialog opens.
4. In the Project Name text field, enter the name of the project (I used `Java21`). The Project Folder field is updated as you type the name. Make a note of this folder—it's where your Java programs can be found on your computer.
5. Deselect the check box Create Main Class.
6. Click Finish.

The project is created. You can use it throughout the lectures for the programs you work on.

If you created a project earlier, it probably will be open in NetBeans. (If not, choose the menu command File, Open Recent Project to select it.) A new class you create will be added to this project.

To begin your first class, run NetBeans and start a new program:

1. Choose the menu command File, New File. The New File dialog opens.
2. In the Categories pane, choose Java.
3. In the File Types pane, choose Empty Java File and click Next. The Empty Java File dialog opens.
4. In the Class Name text field, enter `MarsRobot`. The file you're creating is shown in the Created File field, which can't be edited. This file has the name `MarsRobot.java`.
5. Click Finish.

The NetBeans source code editor opens with nothing in it. Fill it with the code in Listing 1.1. When you're done, save the file using the menu command File, Save. The file `MarsRobot.java` will be saved.

Note

Don’t type the numbers at the beginning of each line in the listing. They’re not part of the program. They are included so that individual lines can be described for instructive purposes in this book.

LISTING 1.1 The Full Text of `MarsRobot.java`.

```
 1: class MarsRobot {
 2:     String status;
 3:     int speed;
 4:     float temperature;
 5:
 6:     void checkTemperature() {
 7:         if (temperature < -80) {
 8:             status = “returning home”;
 9:             speed = 5;
10:         }
11:     }
12:
13:     void showAttributes() {
14:         System.out.println(“Status: ” + status);
15:         System.out.println(“Speed: ” + speed);
16:         System.out.println(“Temperature: ” + temperature);
17:     }
18: }
```

When you save this file, if it has no errors, NetBeans automatically creates a `MarsRobot` class. This process is called *compiling* the class, and it uses a tool called a compiler. The compiler turns the lines of source code into bytecode that the Java Virtual Machine can run.

The `class` statement in line 1 of Listing 1.1 defines and names the `MarsRobot` class. Everything contained between the opening brace `{` on line 1 and the closing brace `}` on line 18 is part of this class.

The `MarsRobot` class contains three instance variables and two instance methods.

The instance variables are defined in lines 2–4:

```
String status;
int speed;
float temperature;
```

The variables are named `status`, `speed`, and `temperature`. Each is used to store a different type of information:

- `status` holds a `String` object—a group of letters, numbers, punctuation, and other characters.
- `speed` holds an `int`, a numeric integer value.
- `temperature` holds a `float`, a floating-point number.

String objects are created from the String class, which is part of the Java Class Library.

Tip

As you might have noticed from the use of String in this program, a class can use an object as an instance variable.

The first instance method in the MarsRobot class is defined in lines 6–11:

```
void checkTemperature() {
    if (temperature < -80) {
        status = "returning home";
        speed = 5;
    }
}
```

Methods are defined in a manner similar to a class. They begin with a statement that names the method, identifies the type of information the method produces, and defines other things.

The checkTemperature() method is contained within the opening brace on line 6 of Listing 1.1 and the closing brace on line 11. This method can be called on a MarsRobot object to find out its temperature.

This method checks to see whether the object's temperature instance variable has a value less than –80. If it does, two other instance variables are changed:

- The status variable is changed to the text "returning home," indicating that the temperature is too cold, and the robot is heading back to its base.
- The speed is changed to 5. (Presumably, this is as fast as the robot can travel.)

The second instance method, showAttributes(), is defined in lines 13–17:

```
void showAttributes() {
    System.out.println("Status: " + status);
    System.out.println("Speed: " + speed);
    System.out.println("Temperature: " + temperature);
}
```

This method calls the method System.out.println() to display the values of three instance variables, along with some text explaining what each value represents.

If you haven't saved this file yet, choose File, Save. This command is disabled if the file hasn't been changed since the last time you saved it.

Running the Program

Even if you typed the MarsRobot program in Listing 1.1 correctly and compiled it into a class, you can't do anything with it. The class you have created defines what a MarsRobot object is like, but it doesn't actually create one of these objects.

There are two ways to put the `MarsRobot` class to use:

- Create a separate Java program that creates an object belonging to that class.
- Add a special class method called `main()` to the `MarsRobot` class so that it can be run as an application. Create an object of that class in that method.

The first option is chosen for this exercise.

Listing 1.2 contains the source code for `MarsApplication`, a Java class that creates a `MarsRobot` object, sets its instance variables, and calls methods. Following the same steps as in the preceding listing, create a new Java file in NetBeans and name it `MarsApplication`.

To begin this second class, follow these steps in NetBeans:

1. Choose File, New File from the menu. The New File dialog opens.
2. In the Categories pane, choose Java.
3. In the File Types pane, choose Empty Java File and click Next. The Empty Java File dialog opens.
4. In the Class Name text field, enter `MarsApplication`. The file you're creating is shown in the Created File field and has the name `MarsApplication.java`.
5. Click Finish.

Enter the code shown in Listing 1.2 into the NetBeans source code editor.

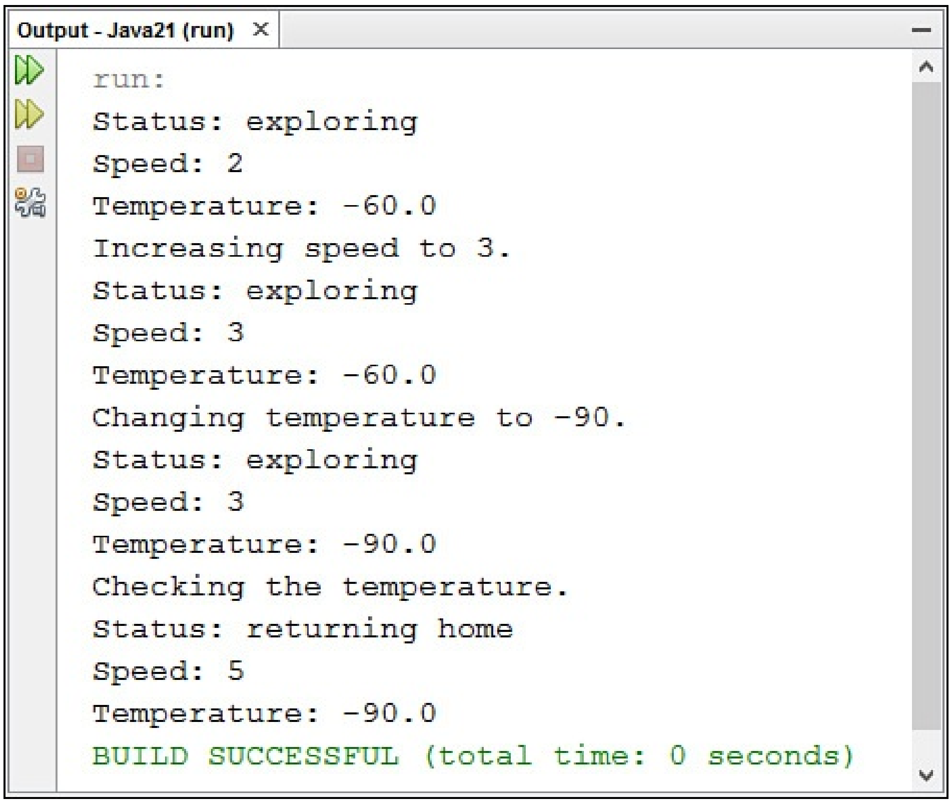
LISTING 1.2 The Full Text of `MarsApplication.java`

```java
class MarsApplication {
    public static void main(String[] arguments) {
        MarsRobot spirit = new MarsRobot();
        spirit.status = "exploring";
        spirit.speed = 2;
        spirit.temperature = -60;

        spirit.showAttributes();
        System.out.println("Increasing speed to 3.");
        spirit.speed = 3;
        spirit.showAttributes();
        System.out.println("Changing temperature to -90.");
        spirit.temperature = -90;
        spirit.showAttributes();
        System.out.println("Checking the temperature.");
        spirit.checkTemperature();
        spirit.showAttributes();
    }
}
```

When you choose File, Save to save the file, NetBeans automatically compiles it into the `MarsApplication` class, which contains bytecode for the JVM to run.

After you have compiled the application, run the program by choosing the menu command Run, Run File. The output displayed by the `MarsApplication` class appears in an Output pane in NetBeans, as shown in Figure 1.1.



```
run:
Status: exploring
Speed: 2
Temperature: -60.0
Increasing speed to 3.
Status: exploring
Speed: 3
Temperature: -60.0
Changing temperature to -90.
Status: exploring
Speed: 3
Temperature: -90.0
Checking the temperature.
Status: returning home
Speed: 5
Temperature: -90.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 1.1 The output of the `MarsApplication` class.

Using Listing 1.2 as a guide, you can see the following things taking place in the `main()` class method of this application:

- **Line 2**—The `main()` method is created and named. All `main()` methods take this format, as you'll see later. " For now,
  the most important thing to note is the `static` keyword, which indicates that the method is a class method shared by all `MarsRobot` objects.
- **Line 3**—A new `MarsRobot` object is created using the class as a template. The object is given the name `spirit`.
- **Lines 4–6**—Three instance variables of the `spirit` object are given values: `status` is set to the text "exploring," `speed` is set to 2, and `temperature` is set to –60.
- **Line 8**—On this line and several that follow, the `showAttributes()` method of the `spirit` object is called. This method displays the current values of the instance variables `status`, `speed`, and `temperature`.

- **Line 9**—On this line and others that follow, a call to the `System.out.println()` method displays the text within parentheses to the output device (your monitor).
- **Line 10**—The `speed` instance variable is set to the value 3.
- **Line 13**—The `temperature` instance variable is set to the value –90.
- **Line 16**—The `checkTemperature()` method of the `spirit` object is called. This method checks to see whether the `temperature` instance variable is less than –80. If it is, `status` and `speed` are assigned new values.

Organizing Classes and Class Behavior

Object-oriented programming in Java also requires three more concepts: inheritance, interfaces, and packages. All three are mechanisms for organizing classes and class behavior.

Inheritance

Inheritance, one of the most crucial concepts in object-oriented programming, has a direct impact on how you design and write your own Java classes.

Inheritance is a mechanism that enables one class to inherit the behavior and attributes of another class.

Through inheritance, a class automatically picks up the functionality of an existing class. The new class must only define how it is different from that existing class.

With inheritance, all classes—including those you create and the ones in the Java Class Library—are arranged in a strict hierarchy.

A class that inherits from another class is called a *subclass*. The class that gives the inheritance is called a *superclass*.

A class can have only one superclass, but it can have an unlimited number of subclasses. Subclasses inherit all the attributes and behavior of their superclass.

In practical terms, this means that if the superclass has behavior and attributes that your class needs, you don’t have to redefine the behavior or copy that code to have the same behavior and attributes. Your class automatically receives these things from its superclass, the superclass gets them from its superclass, and so on, all the way up the hierarchy. Your class becomes a combination of its own features and all the features of the classes above it in the hierarchy.

The situation is comparable to how you inherited traits from your parents, such as your height, hair color, and love of peanut-butter-and-banana sandwiches. They inherited some of these things from their parents, who inherited from theirs, and backward through time to the Garden of Eden, Big Bang, giant spaghetti monster, or *[insert personal belief here]*.

Figure 1.2 shows how a hierarchy of classes is arranged.

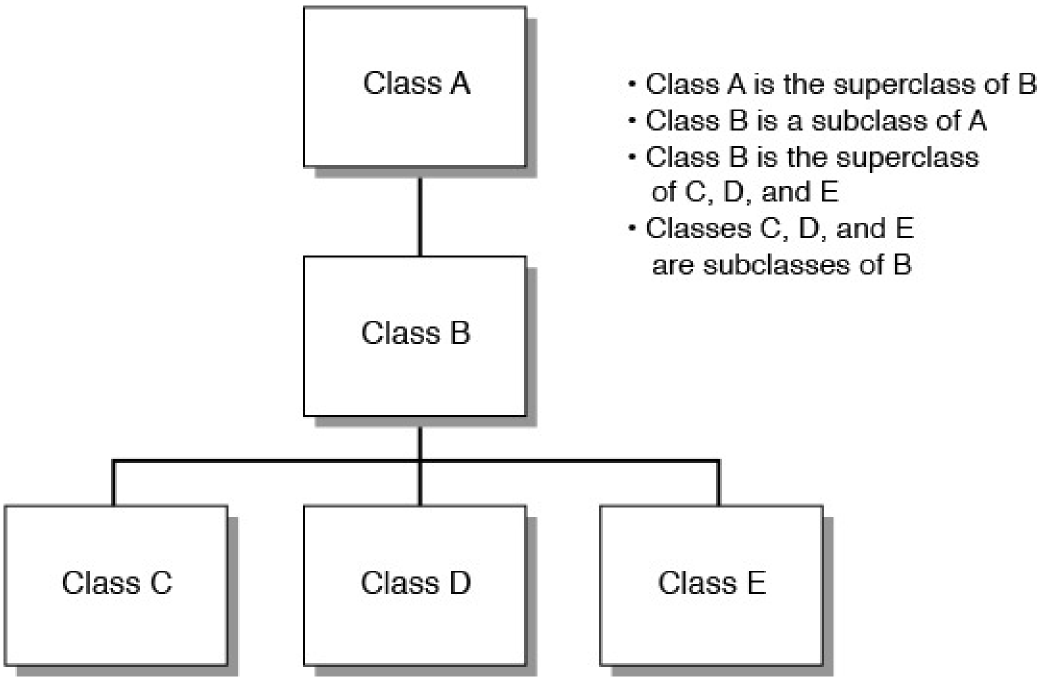

Class A
Class B
Class C
Class D
Class E
• Class A is the superclass of B
• Class B is a subclass of A
• Class B is the superclass of C, D, and E
• Classes C, D, and E are subclasses of B

FIGURE 1.2 A class hierarchy.

At the top of the Java class hierarchy is the class `Object`.

All classes inherit from this superclass. `Object` is the most general class in the hierarchy. It defines behavior inherited by all the classes in the Java Class Library.

Each class further down the hierarchy becomes more tailored to a specific purpose. A class hierarchy defines abstract concepts at the top of the hierarchy. Those concepts become more concrete further down the line of subclasses.

Often when you create a new class in Java, you want all the functionality of an existing class except for some additions or modifications of your own creation. For example, you might want a new version of `CommandButton` that makes a sound when clicked.

To receive all the `CommandButton` functionality without doing any work to re-create it, you can define your new class as a subclass of `CommandButton`.

Because of inheritance, your class automatically inherits behavior and attributes defined in `CommandButton` as well as the behavior and attributes defined in the superclasses of `CommandButton`. All you have to worry about are the things that make your new class different from `CommandButton` itself. Subclassing is the mechanism for defining new classes as the differences between those classes and their superclass.

Subclassing is the creation of a new class that inherits from an existing class. The only task in the subclass is to indicate the differences in behavior and attributes between the subclass and its superclass.

If your class defines entirely new behavior and isn't a subclass of another class, you can inherit directly from the `Object` class.

If you create a class that doesn't indicate a superclass, Java assumes that the new class inherits directly from `Object`. The `MarsRobot` class you created earlier today did not specify a superclass, so it's a subclass of `Object`.

Creating a Class Hierarchy

If you're creating a large set of classes, it makes sense for your classes to inherit from the existing class hierarchy and to make up a hierarchy themselves. This gives your classes several advantages:

- Functionality common to multiple classes can be put into a superclass, which enables it to be used repeatedly in all classes below it in the hierarchy.
- Changes to a superclass automatically are reflected in all its subclasses, their subclasses, and so on. There is no need to change or recompile any of the lower classes; they receive the new information through inheritance.

For example, imagine that you have created a Java class to implement all the features of an exploratory robot. (This shouldn't take much imagination.)

The `MarsRobot` class is completed and works successfully. Your boss at NASA asks you to create a Java class called `MercuryRobot`.

These two kinds of robots have similar features. Both are research robots that work in hostile environments and conduct research. Both keep track of their current temperature and speed.

Your first impulse might be to open the `MarsRobot.java` source file, copy it into a new source file called `MercuryRobot.java`, and then make the necessary changes for the new robot to do its job.

A better plan is to figure out the common functionality of `MercuryRobot` and `MarsRobot` and organize it into a more general class hierarchy. This might be a lot of work just for the classes `MarsRobot` and `MercuryRobot`, but what if you also want to add `MoonRobot`, `UnderseaRobot`, and `DesertRobot`? Factoring common behavior into one or more reusable superclasses significantly reduces the overall amount of work you must do.

To design a class hierarchy that might serve this purpose, start at the top with the class `Object`, the pinnacle of all Java classes.

The most general class to which these robots belong might be called `Robot`. A robot, generally, could be defined as a self-controlled exploration device. In the `Robot` class, you define only the behavior that qualifies something to be a device, to be self-controlled, and to be designed for exploration.

There could be two classes below `Robot`: `WalkingRobot` and `DrivingRobot`. The obvious thing that differentiates these classes is that one travels by foot and the other by wheel. The behavior of walking robots might include bending over to pick up something, ducking, running, and the like. Driving robots would behave differently. Figure 1.3 shows what you have so far.

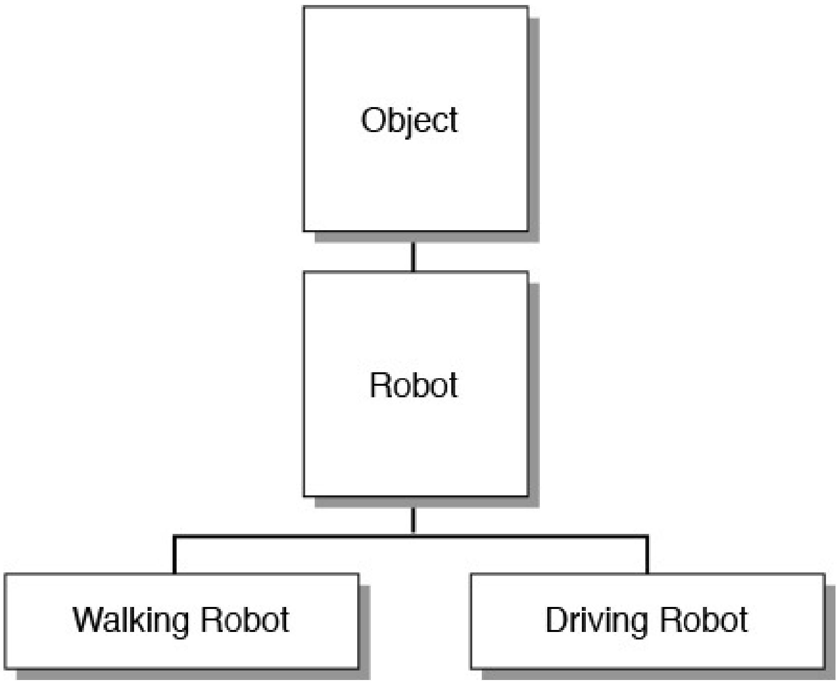


FIGURE 1.3 The basic `Robot` hierarchy.

Now, the hierarchy can become even more specific.

With `WalkingRobot`, you might have several classes: `ScienceRobot`, `GuardRobot`, `SearchRobot`, and so on. As an alternative, you could factor out still more functionality and have intermediate classes for `TwoLegged` and `FourLegged` robots, with different behaviors for each (see Figure 1.4).

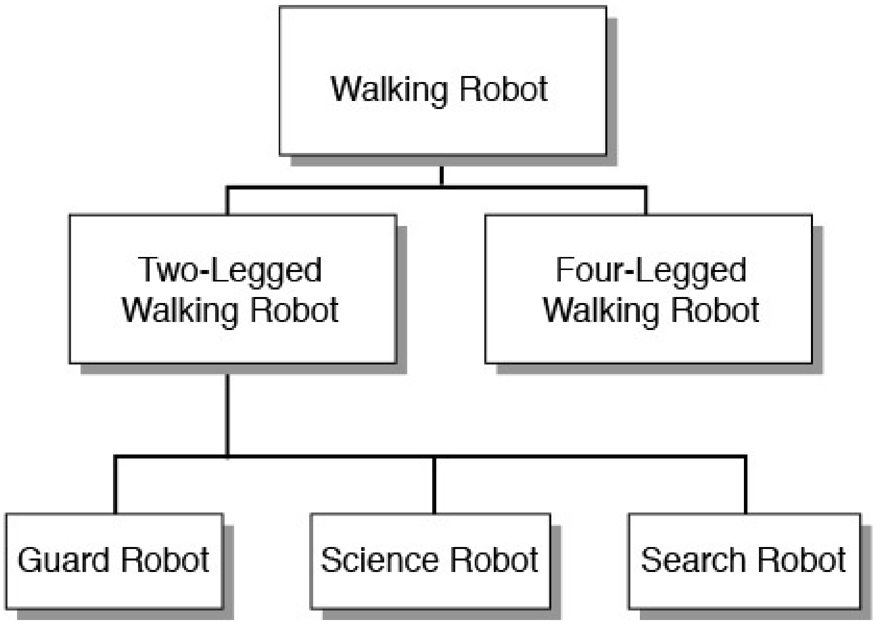


FIGURE 1.4 Two-legged and four-legged walking robots.

Finally, the hierarchy is done, and you have a place for `MarsRobot`. It can be a subclass of `ScienceRobot`, which is a subclass of `WalkingRobot`, which is a subclass of `Robot`, which is a subclass of `Object`.

Where do attributes such as status, temperature, and speed come in? At the place they fit into the class hierarchy most naturally. Because all robots need to keep track of the temperature of their environment, it makes sense to define `temperature` as an instance variable in `Robot`. All subclasses would have that instance variable as well. Remember that you need to define a behavior or attribute only once in the hierarchy, and it is inherited automatically by each subclass.

Note

Designing an effective class hierarchy involves a lot of planning and revision. As you attempt to put attributes and behavior into a hierarchy, you're likely to find reasons to move some classes to different spots in the hierarchy. The goal is to reduce the number of repetitive features (and redundant code) needed.

Inheritance in Action

Inheritance in Java works much more simply than it does in the real world. No wills or courts are required when inheriting from a parent.

When you create a new object, Java keeps track of each variable defined for that object and each variable defined for each superclass of the object. In this way, all the classes combine to form a template for the current object, and each object fills in the information appropriate to its situation.

Methods operate similarly. A new object has access to all method names of its class and superclass. This is determined dynamically when a method is used in a running program. If you call a method of a particular object, the Java virtual machine first checks the object's class for that method. If the method isn't found, the virtual machine looks for it in the superclass of that class, and so on, until the method definition is found. This is illustrated in Figure 1.5.

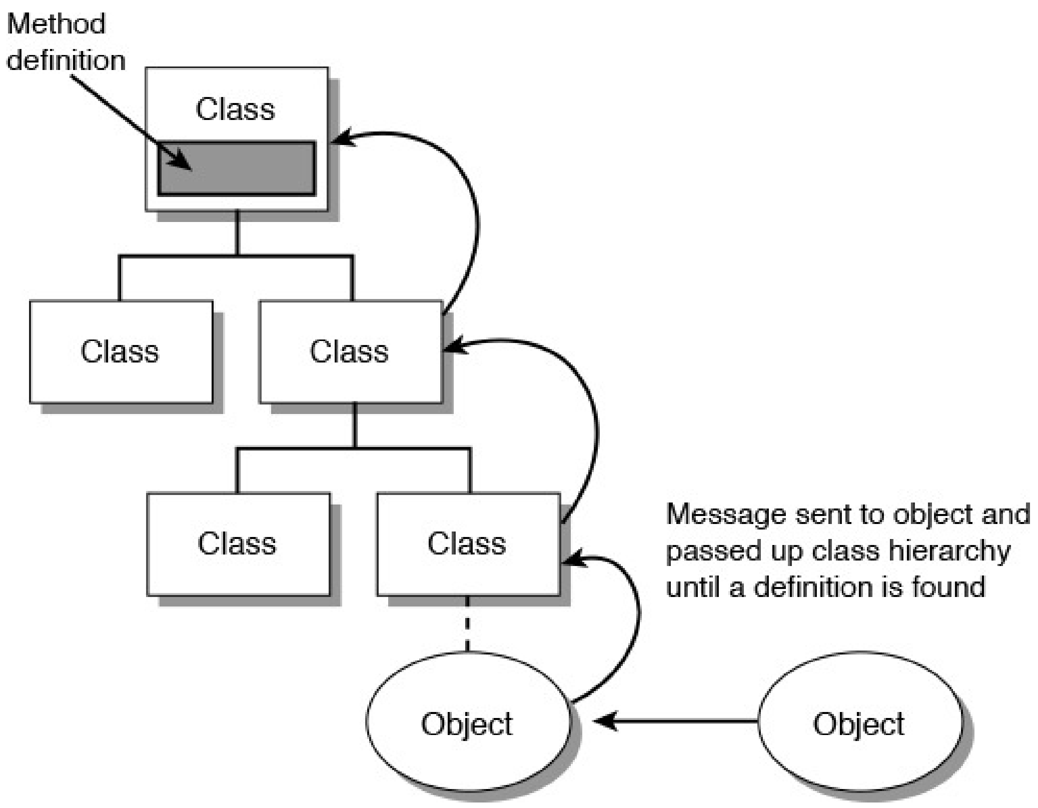

Method definition
Class
Class
Class
Class
Class
Message sent to object and passed up class hierarchy until a definition is found
Object
Object

FIGURE 1.5 How methods are located in a class hierarchy.

Things get complicated when a subclass defines a method that matches a method defined in a superclass in name and other aspects. In this case, the method definition found first (starting at the bottom of the hierarchy and working upward) is the one that is used.

Because of this, you can create a method in a subclass that prevents a method in a superclass from being used. To do this, you give the method the same name, return type, and arguments as the method in the superclass. This procedure, shown in Figure 1.6, is called *overriding*.

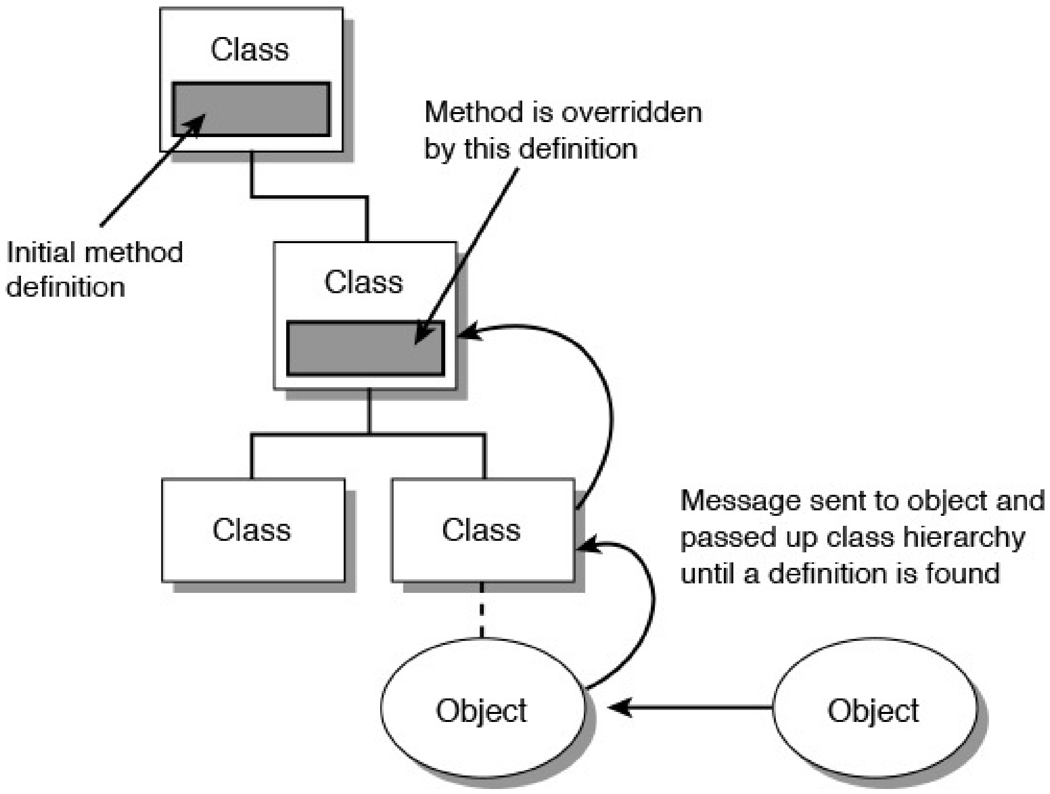

Class
Method is overridden by this definition
Initial method definition
Class
Class
Class
Message sent to object and passed up class hierarchy until a definition is found
Object
Object

FIGURE 1.6 Overriding methods.

Note

Java's form of inheritance is called single inheritance because each Java class can have only one superclass, although any given superclass can have multiple subclasses.

In other object-oriented programming languages such as C++, classes can have more than one superclass, and they inherit combined variables and methods from all those superclasses. This is called multiple inheritance. Java makes inheritance simpler by allowing only single inheritance.

Interfaces

Single inheritance makes the relationship between classes and the functionality they implement easier to understand and design. However, it also can be restrictive, especially when you have similar behavior that needs to be duplicated across different branches of a class hierarchy. Java solves the problem of shared behavior by using interfaces.

An *interface* is a collection of methods that indicate a class has some behavior in addition to what it inherits from its superclasses. The methods included in an interface do not define this behavior; that task is left for the classes that implement the interface.

For example, the `Comparable` interface contains a method that compares two objects of the same class to see which one should appear first in a sorted list. Any class that implements this interface shows other objects that it knows how to determine the sorting order for objects of that class. This behavior would be unavailable to the class without the interface.

You'll learn more about interfaces in the next lectures.

Packages

Packages in Java are a way to group related classes and interfaces. *Packages* enable groups of classes to be referenced more easily in other classes. They also eliminate potential naming conflicts among classes.

Classes in Java can be referred to by a short name such as `Object` or a full name such as `java.lang.Object`.

By default, your Java classes can refer to the classes in the `java.lang` package using only short names. The `java.lang` package provides basic language features such as string handling and mathematical operations. To use classes from any other package, you must refer to them explicitly using their full package name or use an `import` command to import the package in your source code file.

Because the `Color` class is contained in the `java.awt` package, you normally refer to it in your programs with the notation `java.awt.Color`.

If the entire `java.awt` package has been imported using `import`, the class can be referred to as `Color`.

The package for a class is determined by the `package` statement. Many of the classes you create in this book are put in the com.java24hours package, like so:

```
package com.java24hours;
```

This statement must be the first line of the program. When it is omitted, as it was in the `MarsRobot` and `MarsApplication` programs you created today, the class belongs to an unnamed package called the default package.

Summary

If today was your first exposure to object-oriented programming, it probably seemed theoretical and a bit overwhelming.

Because your brain has been stuffed with object-oriented programming concepts and terminology for the first time, you might be worried that no room is left for the remaining java lessons.

Don’t panic. Keep calm and carry on.

At this point, you should have a basic understanding of classes, objects, attributes, and behavior. You also should be familiar with instance variables and methods. You’ll use these right away in the next lecture.

The other aspects of object-oriented programming, such as inheritance and packages, will be covered in more detail in upcoming days.

You’ll work with object-oriented programming in every remaining lectures.
There’s no other way to create programs in Java.

By the time you finish the first lectures, you’ll have working experience with objects, classes, inheritance, and all other aspects of the methodology.

Q&A

Q Methods are functions defined inside classes. If they look like functions and act like functions, why aren't they called functions?

A Some object-oriented programming languages do call them functions. (C++ calls them member functions.) Other object-oriented languages differentiate between functions inside and outside the body of a class or object because in those languages the use of the separate terms is important to understanding how each function works. Because the difference is relevant in other languages and because the term *method* now is in common use in object-oriented terminology, Java uses the term as well.

Q What's the distinction between instance variables and methods and their counterparts, class variables and methods?

A Almost everything you do in a Java program involves instances (also called objects) rather than classes. However, some behavior and attributes make more sense if stored in the class itself rather than in the object.

For example, the `Math` class in the `java.lang` package includes a class variable called `PI` that holds the approximate value of pi. This value does not change, so there's no reason why different objects of that class would need their own individual copy of the `PI` variable. On the other hand, every `String` object contains a method called `length()` that reveals the number of characters in that `String`. This value can be different for each object of that class, so it must be an instance method.

Class variables occupy memory until a Java program is finished running, so they should be used with care. If a class variable references an object, that object will remain in memory as well. This is a common problem causing a program to take up too much memory and run slowly.

Q When a Java class imports an entire package, does it increase the compiled size of that class?

A No. The use of the term "import" is a bit misleading. The `import` keyword does not add the bytecode of one class or one package to the class you are creating. Instead, it makes it easier to refer to classes within another class.

The sole purpose of importing is to shorten the class names when they're used in Java statements. It would be cumbersome to always have to refer to full class names such as `javax.swing.JButton` and `java.awt.Graphics` in your code instead of calling them `JButton` and `Graphics`.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What is another word for a class?

A. Object

B. Template

C. Instance

2. When you create a subclass, what must you define about that class?

A. Nothing. Everything is defined already.

B. Things that are different from its superclass

C. Everything about the class

3. What does an instance method of a class represent?

A. The attributes of that class

B. The behavior of that class

C. The behavior of an object created from that class

Answer it the following questions without looking at today's material.

Which of the following statements is true?

A. All objects created from the same class must be identical.

B. All objects created from the same class can have different attributes than each other.

C. An object inherits attributes and behavior from the class used to create it.

D. A class inherits attributes and behavior from its subclass.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. In the `main()` method of the `MarsRobot` class, create a second `MarsRobot` robot named opportunity, set up its instance variables, and display them.