Java

The Java Programming Language

The Java programming language is a high-level language that can be characterized by all of the following feauters:

- Simple
- Object oriented
- Distributed
- Multithreaded
- Dynamic
- Architecture neutral
- Portable
- High performance
- Robust
- Secure

In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine[1] (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.

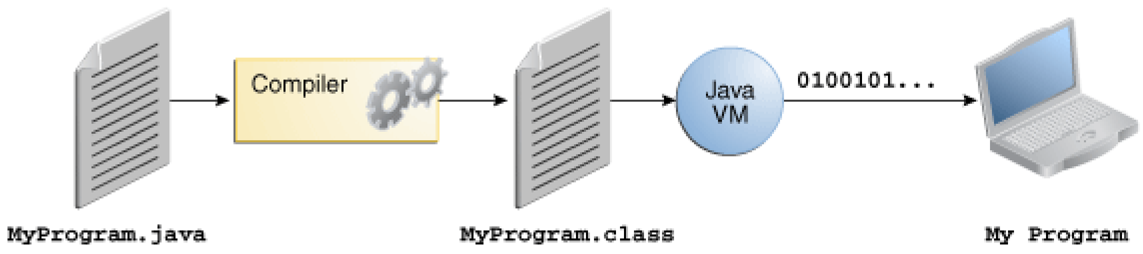


Fig. An overview of the software development process.

Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS. Some virtual machines, such as the Java SE HotSpot at a Glance, perform additional steps at runtime to give your application a performance boost. This include various tasks such as finding performance bottlenecks and recompiling (to native code) frequently used sections of code.

Creating Your First Application

Your first application, HelloWorldApp, will simply display the greeting "Hello world!". To create this program, you will:

- **Create a source file**

A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.

- **Compile the source file into a .class file**

  The Java programming language *compiler* (javac) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.

- **Run the program**

  The Java application *launcher tool* (java) uses the Java virtual machine to run your application.

Create a Source File

To create a source file, you have two options:

- You can save the file HelloWorldApp.java on your computer and avoid a lot of typing. Then, you can go straight to Compile the Source File into a .class File.

First, start your editor. You can launch the Notepad editor from the **Start** menu by selecting **Programs > Accessories > Notepad**. In a new document, type in the following code:

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

- Now compile the file using javac command: javac HelloWorldApp.java
- And execute the class file using java command: java HelloWorldApp
- You will get the HelloWorld! Output

Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—public, or *package-private* (no explicit modifier).
- At the member level—public, private, protected, or *package-private* (no explicit modifier).

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: private and protected. The private modifier specifies that the member can only be accessed in its own class.
The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

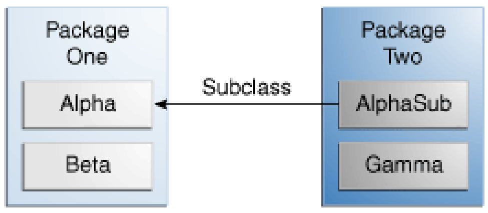
Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.



Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Modifier	**Alpha**	**Beta**	**Alphasub**	**Gamma**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Tips on Choosing an Access Level:

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

The if-then and if-then-else Statements

The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```
void applyBrakes() {
   // the "if" clause: bicycle must be moving
   if (isMoving){
      // the "then" clause: decrease current speed
      currentSpeed--;
   }
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes() {
   // same as above, but without braces
   if (isMoving)
      currentSpeed--;
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in theapplyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes() {
   if (isMoving) {
      currentSpeed--;
   } else {
      System.err.println("The bicycle has " + "already stopped!");
   }
}
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {
   public static void main(String[] args) {

      int testscore = 76;
      char grade;

      if (testscore >= 90) {
         grade = 'A';
      } else if (testscore >= 80) {
         grade = 'B';
      } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

The output from the program is:

```
    Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

The switch Statement

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, andint primitive data types. It also works with *enumerated types* (discussed in Enum Types), the String class, and a few special classes that wrap certain primitive types:Character, Byte, Short, and Integer (discussed in Numbers and Strings).

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code displays the name of the month, based on the value ofmonth, using the switch statement.

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

In this case, August is printed to standard output.

The body of a switch statement is known as a *switch block*. A statement in the switch block can be labeled with one or more case or default labels. The switchstatement evaluates its expression, then executes all statements that follow the matching case label.

You could also display the name of the month with if-then-else statements:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
...  // and so on
```

Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or Stringobject.Another point of interest is the break statement. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following theswitch block. The break statements are necessary because without them, statements in switch blocks *fall through*: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered. The program SwitchDemoFallThrough shows statements in a switch block that fall through. The program displays the month corresponding to the integer month and the months that follow in the year:

```
public class SwitchDemoFallThrough {

    public static void main(String args[]) {
        java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
            case 1:  futureMonths.add("January");
            case 2:  futureMonths.add("February");
            case 3:  futureMonths.add("March");
            case 4:  futureMonths.add("April");
            case 5:  futureMonths.add("May");
            case 6:  futureMonths.add("June");
            case 7:  futureMonths.add("July");
            case 8:  futureMonths.add("August");
            case 9:  futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                     break;
            default: break;
        }

        if (futureMonths.isEmpty()) {
            System.out.println("Invalid month number");
        } else {
            for (String monthName : futureMonths) {
               System.out.println(monthName);
            }
        }
    }
}
```

This is the output from the code:

```
August
September
October
November
December
```

The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
     statement(s)
}
```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in thewhile block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: "
                               + count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){
    // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
     statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the doblock are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: "
                              + count);
          count++;
       } while (count < 11);
    }
}
```

The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination;
     increment) {
    statement(s)
}
```

When using this version of the for statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
   public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: "
                               + i);
        }
   }
}
```

The output of this program is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the forstatement, so it can be used in the termination and increment expressions as well. If the variable that controls a for statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the for loop are optional; an infinite loop can be created as follows:

```
// infinite loop
for ( ; ; ) {

   // your code goes here
}
```

The for statement also has another form designed for iteration through Collections and arrays This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, EnhancedForDemo, uses the enhanced for to loop through the array:

```
class EnhancedForDemo {
   public static void main(String[] args){
       int[] numbers =
          {1,2,3,4,5,6,7,8,9,10};
       for (int item : numbers) {
          System.out.println("Count is: "
                          + item);
       }
   }
}
```

In this example, the variable item holds the current value from the numbers array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

We recommend using this form of the for statement instead of the general form whenever possible.

Branching Statements

The break Statement

The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeledbreak to terminate a for, while, or do-while loop, as shown in the following BreakDemo program:

```
class BreakDemo {
   public static void main(String[] args) {

      int[] arrayOfInts =
         { 32, 87, 3, 589,
          12, 1076, 2000,
          8, 622, 127 };
      int searchfor = 12;
      int i;
      boolean foundIt = false;

      for (i = 0; i < arrayOfInts.length; i++) {
         if (arrayOfInts[i] == searchfor) {
             foundIt = true;
             break;
         }
      }

      if (foundIt) {
         System.out.println("Found " + searchfor + " at index " + i);
      } else {
         System.out.println(searchfor + " not in the array");
      }
   }
}
```

This program searches for the number 12 in an array. The break statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

```
Found 12 at index 4
```

An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement. The following program, BreakWithLabelDemo, is similar to the previous program, but uses nested for loops to search for a value in a two-dimensional array. When the value is found, a labeled break terminates the outer for loop (labeled "search"):

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length;
                 j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor +
                               " at " + i + ", " + j);
        } else {
            System.out.println(searchfor +
                               " not in the array");
        }
    }
}
```

This is the output of the program.Found 12 at 1, 0

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

The continue Statement

The continue statement skips the current iteration of a for, while , or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates theboolean expression that controls the loop. The following program, ContinueDemo , steps through a String, counting the occurences of the letter "p". If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

```
class ContinueDemo {
    public static void main(String[] args) {

        String searchMe
```

```
          = "peter piper picked a " +
            "peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            // process p's
            numPs++;
        }
        System.out.println("Found " +
            numPs + " p's in the string.");
    }
}
```

Here is the output of this program:

```
Found 9 p's in the string.
```

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, ContinueWithLabelDemo, uses the labeled form of continue to skip an iteration in the outer loop.

```
class ContinueWithLabelDemo {
    public static void main(String[] args) {

        String searchMe
           = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() -
                  substring.length();
    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++)
                    != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
                break test;
        }
        System.out.println(foundIt ?
            "Found it" : "Didn't find it");
    }
}
```

Here is the output from this program.

```
Found it
```

The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after thereturn keyword.

```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

```
return;
```

Primitive Data Types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int gear = 1;
```

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other *primitive data types*. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
- **int**: The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.
- **long**: The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by int.
- **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.
- **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in theFloating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

- **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new String object; for example, String s = "this is a string";. String objects are*immutable*, which means that once created, their values cannot be changed. The String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. You'll learn more about the String class in Simple Data Objects

Default Values

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following chart summarizes the default values for the above data types.

Data Type	**Default Value (for fields)**
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

Literals

You may have noticed that the new keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals

An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int. It is recommended that you use the upper case letter L because the lower case letterl is hard to distinguish from the digit 1.

Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long literals. Integer literals can be expressed by these number systems:

- Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
- Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

```
// The number 26, in decimal
int decVal = 26;
//  The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating-Point Literals

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Character and String Literals

Literals of types char and String may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as '\u0108' (capital C with circumflex), or "S\u00ED Se\u00F1or" (Sí Señor in Spanish). Always use 'single quotes' for charliterals and "double quotes" for String literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char orString literals.

The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r(carriage return), \" (double quote), \' (single quote), and \\ (backslash).

There's also a special null literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types. There's little you can do with a null value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal,* formed by taking a type name and appending ".class"; for example, String.class. This refers to the object (of type Class) that represents the type itself.

Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

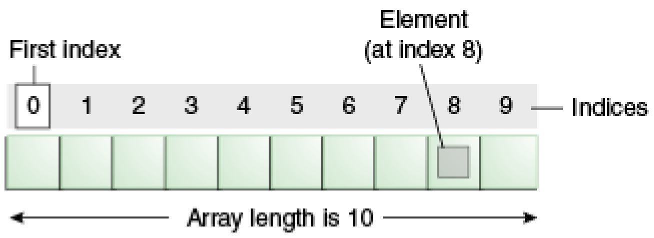
- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// This is an identifier, not
// a numeric literal
int x1 = _52;
// OK (decimal literal)
int x2 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x3 = 52_;
// OK (decimal literal)
int x4 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x5 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x6 = 0x_52;
// OK (hexadecimal literal)
int x7 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x8 = 0x52_;
```

Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You've seen an example of arrays already, in the main method of the "Hello World!" application. This section discusses arrays in greater detail.



An array of ten elements

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

The following program, ArrayDemo, creates an array of integers, puts some values in it, and prints each value to standard output.

```
class ArrayDemo {
  public static void main(String[] args) {
    // declares an array of integers
    int[] anArray;

    // allocates memory for 10 integers
    anArray = new int[10];
```

```
        // initialize first element
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // etc.
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: "
                           + anArray[0]);
        System.out.println("Element at index 1: "
                           + anArray[1]);
        System.out.println("Element at index 2: "
                           + anArray[2]);
        System.out.println("Element at index 3: "
                           + anArray[3]);
        System.out.println("Element at index 4: "
                           + anArray[4]);
        System.out.println("Element at index 5: "
                           + anArray[5]);
        System.out.println("Element at index 6: "
                           + anArray[6]);
        System.out.println("Element at index 7: "
                           + anArray[7]);
        System.out.println("Element at index 8: "
                           + anArray[8]);
        System.out.println("Element at index 9: "
                           + anArray[9]);
    }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
Object dest, int destPos, int length)
```

The two Object arguments specify the array to copy *from* and the array to copy *to*. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, ArrayCopyDemo, declares an array of char elements, spelling the word "decaffeinated". It uses arraycopy to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                          'i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];

      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
   }
}
```

The output from this program is:

```
caffein
```

Classes

The introduction to object-oriented concepts in the lesson titled Object-oriented Programming Concepts used a bicycle class as an example, with racing bikes, mountain bikes, and tandem bikes as subclasses. Here is sample code for a possible implementation of a Bicycle class, to give you an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step. For the moment, don't concern yourself with the details.

```
public class Bicycle {

    // the Bicycle class has
    // three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has
    // one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has
    // four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {
    // the MountainBike subclass has
    // one field
    public int seatHeight;

    // the MountainBike subclass has
    // one constructor
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass has
    // one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it (mountain bikes have seats that can be moved up and down as the terrain demands).

Declaring Classes

You've seen classes defined in the following way:

```
class MyClass {
    // field, constructor, and
    // method declarations
}
```

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

```
class MyClass extends MySuperClass implements YourInterface {
    // field, constructor, and
    // method declarations
}
```

means that MyClass is a subclass of MySuperClass and that it implements the YourInterface interface.

You can also add modifiers like *public* or *private* at the very beginning—so you can see that the opening line of a class declaration can become quite complicated. The modifiers *public* and *private*, which determine what other classes can access MyClass, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the *extends* and *implements* keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, {}.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:

```
public int cadence;
public int gear;
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.

3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only public and private. Other access modifiers will be discussed later.

- public modifier—the field is accessible from all classes.
- private modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

```
public class Bicycle {

   private int cadence;
   private int gear;
   private int speed;

   public Bicycle(int startCadence, int startSpeed, int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }

   public int getCadence() {
      return cadence;
   }

   public void setCadence(int newValue) {
      cadence = newValue;
   }

   public int getGear() {
      return gear;
   }
```

```
    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getSpeed() {
        return speed;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

Types

All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

Defining Methods

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.

4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

The signature of the method declared above is:

calculateAnswer(double, int, double, double)

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
   gear = 1;
   cadence = 10;
   speed = 0;
}
```

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which *does* have a no-argument constructor.

You can use a superclass constructor yourself. The MountainBike class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Note: If another class cannot call a MyClass constructor, it cannot directly create MyClass objects.

Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. For example, the following is a method that computes the monthly payments for a home loan, based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan:

```
public double computePayment(
              double loanAmt,
              double rate,
              double futureValue,
              int numPeriods) {
   double interest = rate / 100.0;
   double partial1 = Math.pow((1 + interest),
               - numPeriods);
   double denominator = (1 - partial1) / interest;
   double answer = (-loanAmt / denominator)
               - ((futureValue * partial1) / denominator);
   return answer;
}
```

This method has four parameters: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

Note: *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers, as you saw in thecomputePayment method, and reference data types, such as objects and arrays.

Here's an example of a method that accepts an array as an argument. In this example, the method creates a new Polygon object and initializes it from an array of Point objects (assume that Point is a class that represents an x, y coordinate):

```
public Polygon polygonFrom(Point[] corners) {
    // method body goes here
}
```

Note: The Java programming language doesn't let you pass methods into methods. But you can pass an object into a method and then invoke the object's methods.

Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following Circle class and its setOrigin method:

```
public class Circle {
    private int x, y, radius;
    public void setOrigin(int x, int y) {
        ...
    }
}
```

The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names x or y within the body of the method refers to the parameter, *not* to the field. To access the field, you must use a qualified name. This will be discussed later in this lesson in the section titled "Using the this Keyword."

Passing Primitive Data Type Arguments

Primitive arguments, such as an int or a double, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;

        // invoke passMethod() with
        // x as argument
        passMethod(x);

        // print x to see if its
        // value has changed
        System.out.println("After invoking passMethod, x = " + x);

    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

When you run this program, the output is:

```
After invoking passMethod, x = 3
```

Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

For example, consider a method in an arbitrary class that moves Circle objects:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of
    // circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new
    // reference to circle
    circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56)
```

Inside the method, circle initially refers to myCircle. The method changes the x and y coordinates of the object that circle references (i.e., myCircle) by 23 and 56, respectively. These changes will persist when the method returns. Then circle is assigned a reference to a new Circle object with x = y = 0. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by circle has changed, but, when the method returns, myCircle still references the same Circle object as before the method was called.

Wrapper Classes

- Wrapper classes are used to convert primitive types to objects, so that the java program will become 100% object oriented, using primitive types you can do only arithmetic operation
- But if we convert that into wrapper you can convert a number to string, octal, hexadecimal etc. which you can’t achieve in primitive types
- List of Wrapper classes
  Byte
  Short
  Integer
  Long
  Float
  Double
  Character
  Boolean

Static Variables and Methods

- Static variables and methods can be accessed using class names
  - For e.g., if you have a global variable public static int a = 20; inside the class Abc
  - then you can access the variable in another class using the class name as Abc.a

- Static methods can be accessed using class name as well

Final methods , variables and classes

- In java we have a keyword called final, which you can use it for classes, methods and variables
- If you use it for variables then you cannot modify that variable
- If you use it for methods then you cannot override that method
- If you use it for classes then you cannot inherit that class
  - final int i = 20;   // cannot be modified
  - final void disp() { }   // cannot be overridden
  - final class Xyz{ }   // cannot be inherited

Abstract methods and classes

- abstract is a keyword you can use it for classes and methods
- if you use it for classes you cannot instantiate the class
- if you use it for the methods then you cannot define the method which will have an abstract keyword you will be writing the declaration so the subclass will provide body for the abstract methods

Interfaces

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know *how* the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

Interfaces in Java

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {

   // constant declarations, if any

   // method signatures

   // An enum with values RIGHT, LEFT
   int turn(Direction direction,
            double radius,
            double startSpeed,
            double endSpeed);
   int changeLanes(Direction direction,
                   double startSpeed,
                   double endSpeed);
   int signalTurn(Direction direction,
                  boolean signalOn);
   int getRadarFront(double distanceToCar,
                     double speedOfCar);
   int getRadarRear(double distanceToCar,
                    double speedOfCar);
         ......
   // more method signatures
}
```

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

```
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:
    int signalTurn(Direction direction, boolean signalOn) {
        // code to turn BMW's LEFT turn indicator lights on
        // code to turn BMW's LEFT turn indicator lights off
        // code to turn BMW's RIGHT turn indicator lights on
        // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes not
    // visible to clients of the interface
}
```

Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Terminology: Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public,protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.

Note: A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject =
    new OuterClass.StaticNestedClass();
```

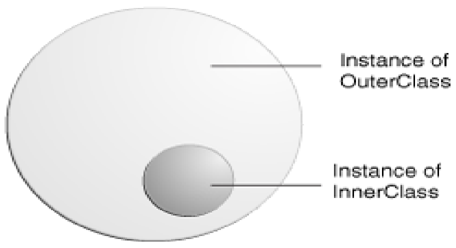
Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass {
   ...
   class InnerClass {
      ...
   }
}
```

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance. The next figure illustrates this idea.



An Instance of InnerClass Exists Within an Instance of OuterClass

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Inner Class Example

To see an inner class in use, let's first consider an array. In the following example, we will create an array, fill it with integer values and then output only values of even indices of the array in ascending order.

The DataStructure class below consists of:

- The DataStructure outer class, which includes methods to add an integer onto the array and print out values of even indices of the array.
- The InnerEvenIterator inner class, which is similar to a standard Java *iterator*. Iterators are used to step through a data structure and typically have methods to test for the last element, retrieve the current element, and move to the next element.
- A main method that instantiates a DataStructure object (ds) and uses it to fill the arrayOfInts array with integer values (0, 1, 2, 3, etc.), then calls a printEvenmethod to print out values of even indices of arrayOfInts.

```
public class DataStructure {
   // create an array
   private final static int SIZE = 15;
   private int[] arrayOfInts = new int[SIZE];

   public DataStructure() {
      // fill the array with ascending integer values
      for (int i = 0; i < SIZE; i++) {
         arrayOfInts[i] = i;
      }
   }

   public void printEven() {
      // print out values of even indices of the array
      InnerEvenIterator iterator = this.new InnerEvenIterator();
      while (iterator.hasNext()) {
         System.out.println(iterator.getNext() + " ");
      }
   }

   // inner class implements the Iterator pattern
   private class InnerEvenIterator {
      // start stepping through the array from the beginning
      private int next = 0;

      public boolean hasNext() {
         // check if a current element is the last in the array
         return (next <= SIZE - 1);
      }

      public int getNext() {
         // record a value of an even index of the array
         int retValue = arrayOfInts[next];
         //get the next even element
```

```
            next += 2;
            return retValue;
        }
    }

    public static void main(String s[]) {
        // fill the array with integer values and print out only
        // values of even indices
        DataStructure ds = new DataStructure();
        ds.printEven();
    }
}
```

The output is:

```
0 2 4 6 8 10 12 14
```

Note that the InnerEvenIterator class refers directly to the arrayOfInts instance variable of the DataStructure object.

Inner classes can be used to implement helper classes like the one shown in the example above. If you plan on handling user-interface events, you will need to know how to use inner classes because the event-handling mechanism makes extensive use of them.

Local and Anonymous Inner Classes

There are two additional types of inner classes. You can declare an inner class within the body of a method. Such a class is known as a *local inner class*. You can also declare an inner class within the body of a method without naming it. These classes are known as *anonymous inner classes*. You will encounter such classes in advanced Java programming.

Modifiers

You can use the same modifiers for inner classes that you use for other members of the outer class. For example, you can use the access specifiers — private, public, and protected — to restrict access to inner classes, just as you do to other class members.

What Is an Exception?

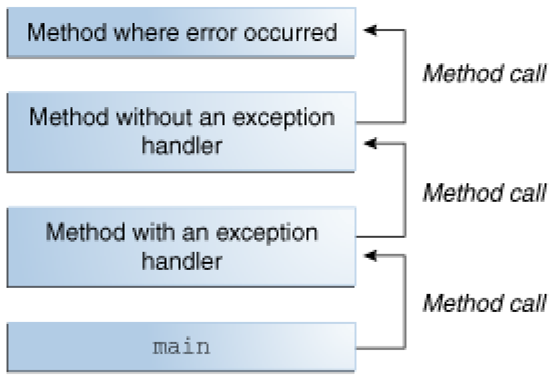
The term *exception* is shorthand for the phrase "exceptional event."

Definition:

An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

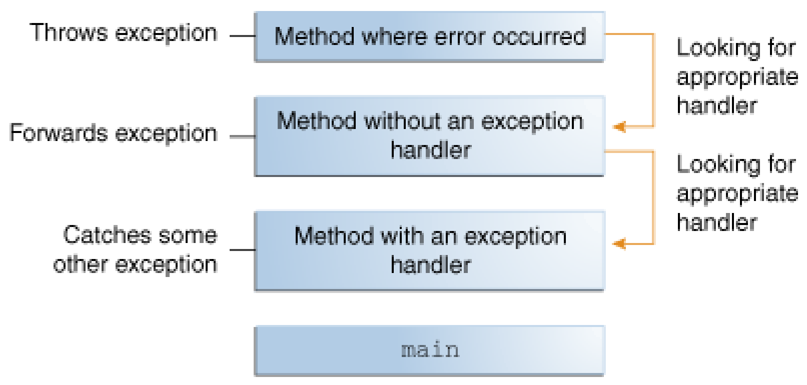
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack* (see the next figure).


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

The call stack.

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

Searching the call stack for the exception handler.

The try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following:

```
try {
    code
}
catch and finally blocks . . .
```

The segment in the example labeled *code* contains one or more legal lines of code that could throw an exception. (The catch and finally blocks are explained in the next two subsections.)

To construct an exception handler for the writeList method from the ListOfNumbers class, enclose the exception-throwing statements of the writeList method within a try block. There is more than one way to do this. You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the writeList code within a single try block and associate multiple handlers with it. The following listing uses one try block for the entire method because the code in question is very short.

```
private List<Integer> list;
private static final int SIZE = 10;

PrintWriter out = null;

try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
}
```
catch and finally statements . . .

The catch Blocks

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

Each catch block is an exception handler and handles the type of exception indicated by its argument. The argument type, *ExceptionType*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class. The handler can refer to the exception with *name*.

The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose *ExceptionType* matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

The following are two exception handlers for the writeList method — one for two types of checked exceptions that can be thrown within the try statement:

```
try {

} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: " + e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

Both handlers print an error message. The second handler does nothing else. By catching any IOException that's not caught by the first handler, it allows the program to continue executing.

The first handler, in addition to printing a message, throws a user-defined exception. (Throwing exceptions is covered in detail later in this chapter in the How to Throw Exceptions section.)

In this example, when the FileNotFoundException is caught it causes a user-defined exception called SampleException to be thrown. You might want to do this if you want your program to handle an exception in this situation in a specific way.

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions, as described in the Chained Exceptions section.

Catching More Than One Type of Exception with One Exception Handler

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Note: If a catch block handles more than one exception type, then the catch parameter is implicitly final. In this example, the catch parameter ex is final and therefore you cannot assign any values to it within the catch block.

The finally Block

The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

Note: If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catchcode is interrupted or killed, the finally block may not execute even though the application as a whole continues.

The try block of the writeList method that you've been working with here opens a PrintWriter. The program should close that stream before exiting the writeListmethod. This poses a somewhat complicated problem because writeList's try block can exit in one of three ways.

1. The new FileWriter statement fails and throws an IOException.
2. The vector.elementAt(i) statement fails and throws an ArrayIndexOutOfBoundsException.
3. Everything succeeds and the try block exits normally.

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

The following finally block for the writeList method cleans up and then closes the PrintWriter.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

In the writeList example, you could provide for cleanup without the intervention of a finally block. For example, you could put the code to close the PrintWriter at the end of the try block and again within the exception handler for ArrayIndexOutOfBoundsException, as follows.

```
try {

    // Don't do this; it duplicates code.
    out.close();

} catch (FileNotFoundException e) {
    // Don't do this; it duplicates code.
    out.close();

    System.err.println("Caught FileNotFoundException: " + e.getMessage());
    throw new RuntimeException(e);
} catch (IOException e) {
   System.err.println("Caught IOException: " + e.getMessage());
}
```

However, this duplicates code, thus making the code difficult to read and error-prone should you modify it later. For example, if you add code that can throw a new type of exception to the try block, you have to remember to close the PrintWriter within the new exception handler.

How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.

As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the Throwable class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

The throw Statement

All methods use the throw statement to throw an exception. The throw statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the Throwable class. Here's an example of a throw statement.

throw *someThrowableObject*;

Let's look at the throw statement in context. The following pop method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object.

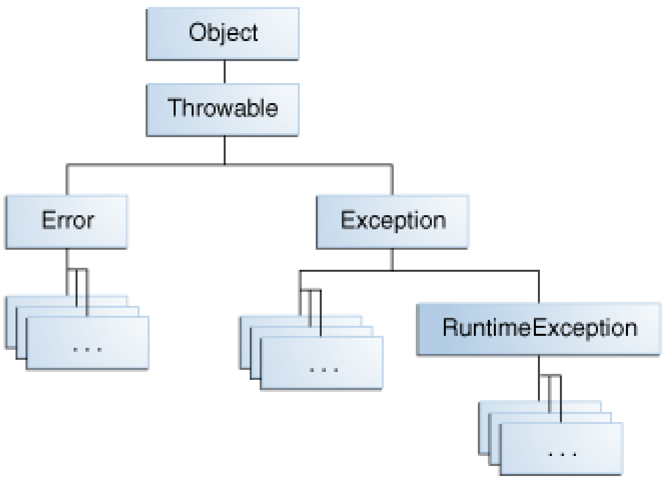
```
public Object pop() {
   Object obj;

   if (size == 0) {
      throw new EmptyStackException();
   }
```

```
    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The pop method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to 0), pop instantiates a new EmptyStackException object (a member of java.util) and throws it.

Throwable Class and Its Subclasses

The objects that inherit from the Throwable class include direct descendants (objects that inherit directly from the Throwable class) and indirect descendants (objects that inherit from children or grandchildren of the Throwable class). The figure below illustrates the class hierarchy of the Throwable class and its most significant subclasses. As you can see, Throwable has two direct descendants: Error and Exception.



The Throwable class.

Error Class

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an Error. Simple programs typically do *not* catch or throwErrors.

Exception Class

Most programs throw and catch objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch Exceptions as opposed to Errors.

The Java platform defines the many descendants of the Exception class. These descendants indicate various types of exceptions that can occur. For example,IllegalAccessException signals that a particular method could not be found, and NegativeArraySizeException indicates that a program attempted to create an array with a negative size.

One Exception subclass, RuntimeException, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception isNullPointerException, which occurs when a method tries to access a member of an object through a null reference.

Specifying the Exceptions Thrown by a Method

The previous section showed how to write an exception handler for the writeList method in the ListOfNumbers class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, if you were providing theListOfNumbers class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to *not* catch the exception and to allow a method further up the call stack to handle it.

If the writeList method doesn't catch the checked exceptions that can occur within it, the writeList method must specify that it can throw these exceptions. Let's modify the original writeList method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the writeList method that won't compile.

```
// Note: This method won't compile by design!
public void writeList() {
   PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
   for (int i = 0; i < SIZE; i++) {
      out.println("Value at: " + i + " = " + vector.elementAt(i));
   }
   out.close();
}
```

To specify that writeList can throw two exceptions, add a throws clause to the method declaration for the writeList method. The throws clause comprises the throwskeyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

public void writeList() **throws IOException, ArrayIndexOutOfBoundsException** {

Remember that ArrayIndexOutOfBoundsException is an unchecked exception; including it in the throws clause is not mandatory. You could just write the following.

```
public void writeList() throws IOException {
//some codes which generates IOException
}
```

Multithreading

Processes and Threads

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object. Multiprocess applications are beyond the scope of this lesson.

Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads

Defining and Starting a Thread

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

- *Provide a Runnable object*. The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:
- public class HelloRunnable implements Runnable {
-
-     public void run() {
-         System.out.println("Hello from a thread!");
-     }
-
-     public static void main(String args[]) {
-         (new Thread(new HelloRunnable())).start();
-     }
-
- }
- *Subclass Thread*. The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:
-
- public class HelloThread extends Thread {
-
-     public void run() {
-         System.out.println("Hello from a thread!");
-     }
-
-     public static void main(String args[]) {
-         (new HelloThread()).start();
-     }
-

- }

Notice that both examples invoke Thread.start in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other thanThread. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread. This lesson focuses on the first approach, which separates the Runnable task from the Thread object that executes the task. Not only is this approach more flexible, but it is applicable to the high-level thread management APIs covered later.

The Thread class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and Thread object.

Pausing Execution with Sleep

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The sleep method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements, as with the SimpleThreads example in a later section.

Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts, as we'll see in a later section. In any case, you cannot assume that invoking sleep will suspend the thread for precisely the time period specified.

The SleepMessages example uses sleep to print messages at four-second intervals:

```
public class SleepMessages {
    public static void main(String args[])
        throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        for (int i = 0;
             i < importantInfo.length;
             i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

Notice that main declares that it throws InterruptedException. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active. Since this application has not defined another thread to cause the interrupt, it doesn't bother to catch InterruptedException

Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the synchronized keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a List called instances containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```
But then other threads can use instances to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.

Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.

Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time. This example application, Deadlock, models this possibility:

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

When Deadlock runs, it's extremely likely that both threads will block when they attempt to invoke bowBack. Neither block will ever end, because each thread is waiting for the other to exit bow.

Thread Interference

Consider a simple class called Counter

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

Counter is designed so that each invocation of increment will add 1 to c, and each invocation of decrement will subtract 1 from c. However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

It might not seem possible for operations on instances of Counter to interleave, since both operations on c are single, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single expression c++ can be decomposed into three steps:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

The expression c-- can be decomposed the same way, except that the second step decrements instead of increments.

Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.

5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

Introduction to Collections

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

If you've used the Java programming language — or just about any other programming language — you're already familiar with collections. Collection implementations in earlier (pre-1.2) versions of the Java platform included Vector, Hashtable, and array. However, those earlier versions did not contain a collections framework.

What Is a Collections Framework?

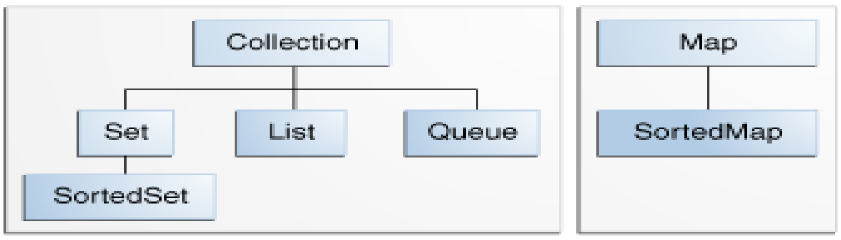
A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java

Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

The Collection Interface

A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a*conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

```
List<String> list = new ArrayList<String>(c);
```

The following shows the Collection interface.

```
public interface Collection<E> extends Iterable<E> {
   // Basic operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   // optional
   boolean add(E element);
   // optional
   boolean remove(Object element);
   Iterator<E> iterator();

   // Bulk operations
   boolean containsAll(Collection<?> c);
   // optional
```

```
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The interface does about what you'd expect given that a Collection represents a group of objects. The interface has methods to tell you how many elements are in the collection (size, isEmpty), to check whether a given object is in the collection (contains), to add and remove an element from the collection (add, remove), and to provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is designed to remove a single instance of the specified element from the Collection, assuming that it contains the element to start with, and to return true if the Collection was modified as a result.

The Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited fromCollection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

The following is the Set interface.

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
// optional
void clear();

// Array Operations
Object[] toArray();
<T> T[] toArray(T[] a);
}
```

The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet. HashSet, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided byHashSet at a cost that is only slightly higher.

The List Interface

A List is an ordered Collection (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the Listinterface includes operations for the following:

- Positional access — manipulates elements based on their numerical position in the list
- Search — searches for a specified object in the list and returns its numerical position
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature
- Range-view — performs arbitrary *range operations* on the list.

The List interface follows.

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    // optional
    E set(int index, E element);
    // optional
    boolean add(E element);
    // optional
    void add(int index, E element);
    // optional
    E remove(int index);
    // optional
    boolean addAll(int index, Collection<? extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

The Java platform contains two general-purpose List implementations. ArrayList, which is usually the better-performing implementation, and LinkedList which offers better performance under certain circumstances. Also, Vector has been retrofitted to implement List.

Comparison to Vector

If you've used Vector, you're already familiar with the general basics of List. (Of course, List is an interface, while Vector is a concrete implementation.) List fixes several minor API deficiencies in Vector. Commonly used Vector operations, such as elementAt and setElementAt, have been given much shorter names. When you consider that these two operations are the List analog of square brackets for arrays, it becomes apparent that shorter names are highly desirable. Consider the following assignment statement.

```
a[i] = a[j].times(a[k]);
```

The Vector equivalent is:

```
v.setElementAt(v.elementAt(j).times(v.elementAt(k)), i);
```

The List equivalent is:

```
v.set(i, v.get(j).times(v.get(k)));
```

You may already have noticed that the set method, which replaces the Vector method setElementAt, reverses the order of the arguments so that they match the corresponding array operation. Consider the following assignment statement.

```
gift[5] = "golden rings";
```

The Vector equivalent is:

```
gift.setElementAt("golden rings", 5);
```

The List equivalent is:

```
gift.set(5, "golden rings");
```

For consistency's sake, the method add(int, E), which replaces insertElementAt(Object, int), also reverses the order of the arguments.

The three range operations in Vector (indexOf, lastIndexOf, and setSize) have been replaced by a single range-view operation (subList), which is far more powerful and consistent.

Iterators

As you'd expect, the Iterator returned by List's iterator operation returns the elements of the list in proper sequence. List also provides a richer iterator, called aListIterator, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator. The ListIteratorinterface follows.

```
public interface ListIterator<E> extends Iterator<E> {
   boolean hasNext();
   E next();
   boolean hasPrevious();
   E previous();
   int nextIndex();
   int previousIndex();
   void remove(); //optional
   void set(E e); //optional
   void add(E e); //optional
}
```

The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and theprevious operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

Here's the standard idiom for iterating backward through a list.

```
for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {
   Type t = it.previous();
   ...
}
```

The Queue Interface

A Queue is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations. The Queue interface follows.

```
public interface Queue<E> extends Collection<E> {
   E element();
   boolean offer(E e);
   E peek();
   E poll();
   E remove();
}
```

Each Queue method exists in two forms: (1) one throws an exception if the operation fails, and (2) the other returns a special value if the operation fails (either null or false, depending on the operation). The regular structure of the interface is illustrated in the following table.

Queue Interface Structure

Type of Operation	**Throws exception**	**Returns special value**
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to their values — see the Object Ordering section for details). Whatever ordering is used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

It is possible for a Queue implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some Queue implementations injava.util.concurrent are bounded, but the implementations in java.util are not.

The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The Map interface follows.

```
public interface Map<K,V> {

   // Basic operations
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // Bulk operations
   void putAll(Map<? extends K, ? extends V> m);
   void clear();

   // Collection Views
   public Set<K> keySet();
   public Collection<V> values();
   public Set<Map.Entry<K,V>> entrySet();

   // Interface for entrySet elements
   public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
   }
}
```

The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet, as described in The Set Interface section. Also, Hashtable was retrofitted to implement Map.

Comparison to Hashtable

If you've used Hashtable, you're already familiar with the general basics of Map. (Of course, Map is an interface, while Hashtable is a concrete implementation.) The following are the major differences:

- Map provides Collection views instead of direct support for iteration via Enumeration objects. Collection views greatly enhance the expressiveness of the interface, as discussed later in this section.
- Map allows you to iterate over keys, values, or key-value pairs; Hashtable does not provide the third option.
- Map provides a safe way to remove entries in the midst of iteration; Hashtable did not.

Finally, Map fixes a minor deficiency in the Hashtable interface. Hashtable has a method called contains, which returns true if the Hashtable contains a given *value*. Given its name, you'd expect this method to return true if the Hashtable contained a given *key*, because the key is the primary access mechanism for a Hashtable. The Mapinterface eliminates this source of confusion by renaming the method containsValue. Also, this improves the interface's consistency — containsValue parallelscontainsKey.

The SortedMap Interface

A SortedMap is a Map that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a Comparator provided at the time of theSortedMap creation. Natural ordering and Comparators are discussed in the Object Ordering section. The SortedMap interface provides operations for normal Mapoperations and for the following:

- Range view — performs arbitrary range operations on the sorted map
- Endpoints — returns the first or the last key in the sorted map
- Comparator access — returns the Comparator, if any, used to sort the map

The following interface is the Map analog of SortedSet.

```
public interface SortedMap<K, V> extends Map<K, V>{
   Comparator<? super K> comparator();
   SortedMap<K, V> subMap(K fromKey, K toKey);
   SortedMap<K, V> headMap(K toKey);
```

```
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Map Operations

The operations SortedMap inherits from Map behave identically on sorted maps and normal maps with two exceptions:

- The Iterator returned by the iterator operation on any of the sorted map's Collection views traverse the collections in order.
- The arrays returned by the Collection views' toArray operations contain the keys, values, or entries in order.

Although it isn't guaranteed by the interface, the toString method of the Collection views in all the Java platform's SortedMap implementations returns a string containing all the elements of the view, in order.

I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

Byte Streams

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream.

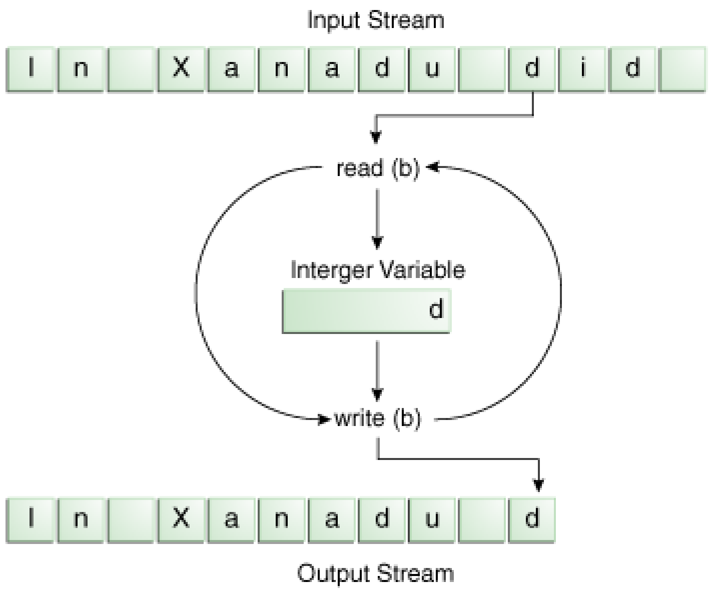
There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Using Byte Streams

We'll explore FileInputStream and FileOutputStream by examining an example program named CopyBytes, which uses byte streams to copy xanadu.txt, one byte at a time.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Simple byte stream input and output.

Notice that read() returns an int value. If the input is a stream of bytes, why doesn't read() return a byte value? Using a int as a return type allows read() to use -1 to indicate that it has reached the end of the stream.

Always Close Streams

Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that CopyBytes was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial nullvalue. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

When Not to Use Byte Streams

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since xanadu.txt contains character data, the best approach is to use character streams, as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

Character Streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding. See the Internationalization trail for more information.

Using Character Streams

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader andFileWriter. The CopyCharacters example illustrates these classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place ofFileInputStream and FileOutputStream.

Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, inCopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: InputStreamReader and OutputStreamWriter. Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, BufferedReader and PrintWriter. We'll explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

The CopyLines example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Invoking readLine returns a line of text with the line. CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

Buffered Streams

Most of the examples we've seen so far use *unbuffered* I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the CopyCharacters example to use buffered I/O:

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, whileBufferedReader and BufferedWriter create buffered character streams.

Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer.

Some buffered output classes support *autoflush*, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format.

To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered. Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface Serializable.

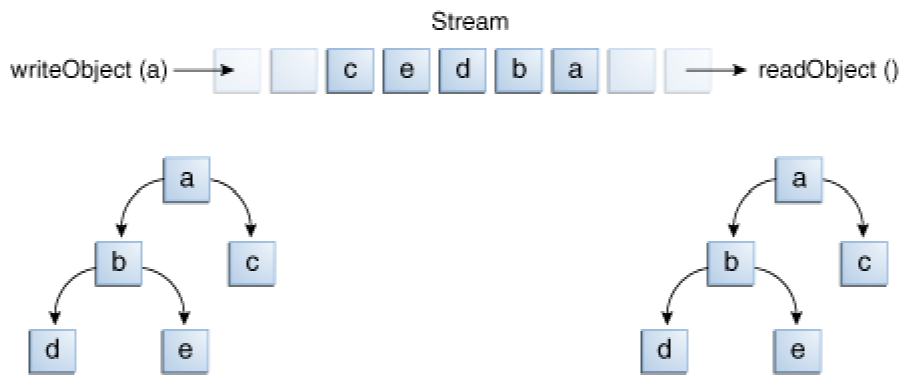
The object stream classes are ObjectInputStream and ObjectOutputStream. These classes implement ObjectInput and ObjectOutput, which are subinterfaces ofDataInput and DataOutput. That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values. The ObjectStreams example illustrates this. ObjectStreams creates the same application as DataStreams, with a couple of changes. First, prices are now BigDecimalobjects, to better represent fractional values. Second, a Calendar object is written to the data file, indicating an invoice date.

If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a ClassNotFoundException. In this simple example, that can't happen, so we don't try to catch the exception. Instead, we notify the compiler that we're aware of the issue by adding ClassNotFoundException to the main method'sthrows clause.

Output and Input of Complex Objects

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObjecttraverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where writeObject is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b**contains references to **d** and **e**. Invoking writeobject(a) writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object ob twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

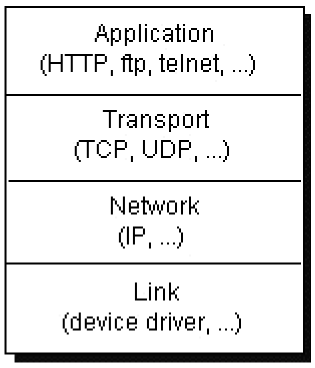
```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

This results in two variables, ob1 and ob2, that are references to a single object.

Networking

Networking Basics

Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP), as this diagram illustrates:



When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the java.net package. These classes provide system-independent network communication. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

TCP

When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection. This is analogous to making a telephone call. If you want to speak to Aunt Beatrice in Kentucky, a connection is established when you dial her phone number and she answers. You send data back and forth over the connection by speaking to one another over the phone lines. Like the phone company, TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.

TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

Definition:

TCP (*Transmission Control Protocol*) is a connection-based protocol that provides a reliable flow of data between two computers.

UDP

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other.

Definition:

UDP (*User Datagram Protocol*) is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival. UDP is not connection-based like TCP.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.

What Is a URL?

If you've been surfing the Web, you have undoubtedly heard the term URL and have used URLs to access HTML pages from the Web.

It's often easiest, although not entirely accurate, to think of a URL as the name of a file on the World Wide Web because most URLs refer to a file on some machine on the network.

However, remember that URLs also can point to other resources on the network, such as database queries and command output.

Definition:

URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.

A URL has two main components:

- Protocol identifier: For the URL http://example.com, the protocol identifier is http.
- Resource name: For the URL http://example.com, the resource name is example.com.

Note that the protocol identifier and the resource name are separated by a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:

Host Name

The name of the machine on which the resource lives.

Filename

The pathname to the file on the machine.

Port Number

The port number to which to connect (typically optional).

Reference

A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

Creating a URL

The easiest way to create a URL object is from a String that represents the human-readable form of the URL address. This is typically the form that another person will use for a URL. In your Java program, you can use a String containing this text to create a URL object:

```
URL myURL = new URL("http://example.com/");
```

The URL object created above represents an *absolute URL*. An absolute URL contains all of the information necessary to reach the resource in question. You can also createURL objects from a *relative URL* address.

Creating a URL Relative to Another

A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL.

Relative URL specifications are often used within HTML files. For example, suppose you write an HTML file called JoesHomePage.html. Within this page, are links to other pages, PicturesOfMe.html and MyKids.html, that are on the same machine and in the same directory as JoesHomePage.html. The links to PicturesOfMe.html andMyKids.html from JoesHomePage.html could be specified just as filenames, like this:

```
<a href="PicturesOfMe.html">Pictures of Me</a>
<a href="MyKids.html">Pictures of My Kids</a>
```

These URL addresses are *relative URLs*. That is, the URLs are specified relative to the file in which they are contained — JoesHomePage.html.

In your Java programs, you can create a URL object from a relative URL specification. For example, suppose you know two URLs at the site example.com:

```
http://example.com/pages/page1.html
http://example.com/pages/page2.html
```

You can create URL objects for these pages relative to their common base URL: http://example.com/pages/ like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:

URL(URL *baseURL*, String *relativeURL*)

The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base. IfbaseURL is null, then this constructor treats relativeURL like an absolute URL specification. Conversely, if relativeURL is an absolute URL specification, then the constructor ignores baseURL.

This constructor is also useful for creating URL objects for named anchors (also called references) within a file. For example, suppose the page1.html file has a named anchor called BOTTOM at the bottom of the file. You can use the relative URL constructor to create a URL object for it like this:

```
URL page1BottomURL = new URL(page1URL,"#BOTTOM");
```

Connecting to a URL

After you've successfully created a URL object, you can call the URL object's openConnection method to get a URLConnection object, or one of its protocol specific subclasses, e.g. java.net.HttpURLConnection

You can use this URLConnection object to setup parameters and general request properties that you may need before connecting. Connection to the remote object represented by the URL is only initiated when the URLConnection.connect method is called. When you do this you are initializing a communication link between your Java program and the URL over the network. For example, the following code opens a connection to the site example.com:

```
try {
   URL myURL = new URL("http://example.com/");
   URLConnection myURLConnection = myURL.openConnection();
   myURLConnection.connect();
}
catch (MalformedURLException e) {
   // new URL() failed
   // ...
}
catch (IOException e) {
   // openConnection() failed
   // ...
}
```

A new URLConnection object is created every time by calling the openConnection method of the protocol handler for this URL.

Reading from and Writing to a URLConnection

The URLConnection class contains many methods that let you communicate with the URL over the network. URLConnection is an HTTP-centric class; that is, many of its methods are useful only when you are working with HTTP URLs. However, most URL protocols allow you to read from and write to the connection. This section describes both functions.

Reading from a URLConnection

The following program performs the same function as the URLReader program shown in Reading Directly from a URL.

However, rather than getting an input stream directly from the URL, this program explicitly retrieves a URLConnection object and gets an input stream from the connection. The connection is opened implicitly by calling getInputStream. Then, like URLReader, this program creates a BufferedReader on the input stream and reads from it. The bold statements highlight the differences between this example and the previous:

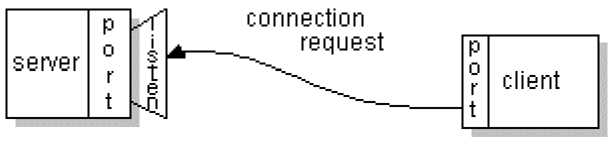
```
import java.net.*;
import java.io.*;

public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL oracle = new URL("http://www.oracle.com/");
        URLConnection yc = oracle.openConnection();
        BufferedReader in = new BufferedReader(new InputStreamReader(
                                    yc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

The output from this program is identical to the output from the program that opens a stream directly from the URL. You can use either way to read from a URL. However, reading from a URLConnection instead of reading directly from a URL might be more useful. This is because you can use the URLConnection object for other tasks (like writing to the URL) at the same time.

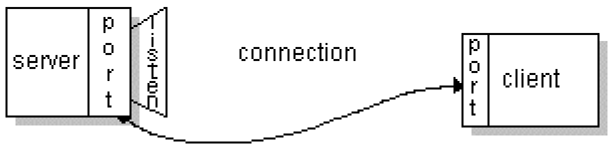
What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using thejava.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, java.net includes the ServerSocket class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the Socket and ServerSocket classes.

If you are trying to connect to the Web, the URL class and related classes (URLConnection, URLEncoder) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation. See Working with URLs for information about connecting to the Web via URLs.

Reading from and Writing to a Socket

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the Socket class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client, EchoClient, that connects to the Echo server. The Echo server simply receives data from its client and echoes it back. The Echo server is a well-known service that clients can rendezvous with on port 7.

EchoClient creates a socket thereby getting a connection to the Echo server. It reads input from the user on the standard input stream, and then forwards that text to the Echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server:

```
import java.io.*;
import java.net.*;

public class EchoClient {
    public static void main(String[] args) throws IOException {

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket("taranis", 7);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                                        echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                               + "the connection to: taranis.");
            System.exit(1);
        }

              BufferedReader stdIn = new BufferedReader(
                                   new InputStreamReader(System.in));
              String userInput;

              while ((userInput = stdIn.readLine()) != null) {
                  out.println(userInput);
                  System.out.println("echo: " + in.readLine());
              }

              out.close();
              in.close();
              stdIn.close();
              echoSocket.close();
    }
}
```

Note that EchoClient both writes to and reads from its socket, thereby sending data to and receiving data from the Echo server.

Let's walk through the program and investigate the interesting parts. The three statements in the try block of the main method are critical. These lines establish the socket connection between the client and the server and open a PrintWriter and a BufferedReader on the socket:

```
echoSocket = new Socket("taranis", 7);
out = new PrintWriter(echoSocket.getOutputStream(), true);
in = new BufferedReader( new InputStreamReader(echoSocket.getInputStream()));
```

The first statement in this sequence creates a new Socket object and names it echoSocket. The Socket constructor used here requires the name of the machine and the port number to which you want to connect. The example program uses the host name taranis. This is the name of a hypothetical machine on our local network. When you type in and run this program on your machine, change the host name to the name of a machine on your network. Make sure that the name you use is the fully qualified IP name of the machine to which you want to connect. The second argument is the port number. Port number 7 is the port on which the Echo server listens.

The second statement gets the socket's output stream and opens a PrintWriter on it. Similarly, the third statement gets the socket's input stream and opens aBufferedReader on it. The example uses readers and writers so that it can write Unicode characters over the socket.

To send data through the socket to the server, EchoClient simply needs to write to the PrintWriter. To get the server's response, EchoClient reads from theBufferedReader. The rest of the program achieves this. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the while loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the PrintWriter connected to the socket:

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
   out.println(userInput);
   System.out.println("echo: " + in.readLine());
}
```

The last statement in the while loop reads a line of information from the BufferedReader connected to the socket. The readLine method waits until the server echoes the information back to EchoClient. When readline returns, EchoClient prints the information to the standard output.

The while loop continues until the user types an end-of-input character. That is, EchoClient reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of-input. The while loop then terminates and the program continues, executing the next four lines of code:

```
out.close();
in.close();
stdIn.close();
echoSocket.close();
```

These lines of code fall into the category of housekeeping. A well-behaved program always cleans up after itself, and this program is well-behaved. These statements close the readers and writers connected to the socket and to the standard input stream, and close the socket connection to the server. The order here is important. You should close any streams connected to a socket before you close the socket itself.

This client program is straightforward and simple because the Echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.

Writing the Server Side of a Socket

This section shows you how to write a server and the client that goes with it. The server in the client/server pair serves up Knock Knock jokes. Knock Knock jokes are favored by children and are usually vehicles for bad puns. They go like this:

Server: "Knock knock!"
Client: "Who's there?"
Server: "Dexter."
Client: "Dexter who?"
Server: "Dexter halls with boughs of holly."
Client: "Groan."

The example consists of two independently running Java programs: the client program and the server program. The client program is implemented by a single class,KnockKnockClient, and is very similar to the EchoClient example from the previous section. The server program is implemented by two classes: KnockKnockServer andKnockKnockProtocol, KnockKnockServer contains the main method for the server program and performs the work of listening to the port, establishing connections, and reading from and writing to the socket. KnockKnockProtocol serves up the jokes. It keeps track of the current joke, the current state (sent knock knock, sent clue, and so on), and returns the various text pieces of the joke depending on the current state. This object implements the protocol-the language that the client and server have agreed to use to communicate.

The following section looks in detail at each class in both the client and the server and then shows you how to run them.

The Knock Knock Server

This section walks through the code that implements the Knock Knock server program. Here is the complete source for the KnockKnockServer class.

The server program begins by creating a new ServerSocket object to listen on a specific port (see the statement in bold in the following code segment). When writing a server, choose a port that is not already dedicated to some other service. KnockKnockServer listens on port 4444 because 4 happens to be my favorite number and port 4444 is not being used for anything else in my environment:

```
try {
   serverSocket = new ServerSocket(4444);
}
catch (IOException e) {
   System.out.println("Could not listen on port: 4444");
   System.exit(-1);
}
```

ServerSocket is a java.net class that provides a system-independent implementation of the server side of a client/server socket connection. The constructor forServerSocket throws an exception if it can't listen on the specified port (for example, the port is already being used). In this case, the KnockKnockServer has no choice but to exit.

If the server successfully binds to its port, then the ServerSocket object is successfully created and the server continues to the next step--accepting a connection from a client (shown in bold):

```
Socket clientSocket = null;
try {
   clientSocket = serverSocket.accept();
}
catch (IOException e) {
   System.out.println("Accept failed: 4444");
   System.exit(-1);
}
```

The accept method waits until a client starts up and requests a connection on the host and port of this server (in this example, the server is running on the hypothetical machine taranis on port 4444). When a connection is requested and successfully established, the accept method returns a new Socket object which is bound to the same local port and has its remote address and remote port set to that of the client. The server can communicate with the client over this new Socket and continue to listen for client connection requests on the original ServerSocket This particular version of the program doesn't listen for more client connection requests. However, a modified version of the program is provided in Supporting Multiple Clients.

After the server successfully establishes a connection with a client, it communicates with the client using this code:

```
PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
BufferedReader in =
   new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
String inputLine, outputLine;

// initiate conversation with client
KnockKnockProtocol kkp = new KnockKnockProtocol();
outputLine = kkp.processInput(null);
out.println(outputLine);

while ((inputLine = in.readLine()) != null) {
   outputLine = kkp.processInput(inputLine);
   out.println(outputLine);
   if (outputLine.equals("Bye."))
   break;
}
```

This code:

1. Gets the socket's input and output stream and opens readers and writers on them.
2. Initiates communication with the client by writing to the socket (shown in bold).
3. Communicates with the client by reading from and writing to the socket (the while loop).

Step 1 is already familiar. Step 2 is shown in bold and is worth a few comments. The bold statements in the code segment above initiate the conversation with the client. The code creates a KnockKnockProtocol object-the object that keeps track of the current joke, the current state within the joke, and so on.

After the KnockKnockProtocol is created, the code calls KnockKnockProtocol's processInput method to get the first message that the server sends to the client. For this example, the first thing that the server says is "Knock! Knock!" Next, the server writes the information to the PrintWriter connected to the client socket, thereby sending the message to the client.

Step 3 is encoded in the while loop. As long as the client and server still have something to say to each other, the server reads from and writes to the socket, sending messages back and forth between the client and the server.

The server initiated the conversation with a "Knock! Knock!" so afterwards the server must wait for the client to say "Who's there?" As a result, the while loop iterates on a read from the input stream. The readLine method waits until the client responds by writing something to its output stream (the server's input stream). When the client responds, the server passes the client's response to the KnockKnockProtocol object and asks the KnockKnockProtocol object for a suitable reply. The server immediately sends the reply to the client via the output stream connected to the socket, using a call to println. If the server's response generated from the KnockKnockServer object is "Bye." this indicates that the client doesn't want any more jokes and the loop quits.

The KnockKnockServer class is a well-behaved server, so the last several lines of this section of KnockKnockServer clean up by closing all of the input and output streams, the client socket, and the server socket:

```
out.close();
in.close();
clientSocket.close();
serverSocket.close();
```

The Knock Knock Protocol

The KnockKnockProtocol class implements the protocol that the client and server use to communicate. This class keeps track of where the client and the server are in their conversation and serves up the server's response to the client's statements. The KnockKnockServer object contains the text of all the jokes and makes sure that the client gives the proper response to the server's statements. It wouldn't do to have the client say "Dexter who?" when the server says "Knock! Knock!"

All client/server pairs must have some protocol by which they speak to each other; otherwise, the data that passes back and forth would be meaningless. The protocol that your own clients and servers use depends entirely on the communication required by them to accomplish the task.

The Knock Knock Client

The KnockKnockClient class implements the client program that speaks to the KnockKnockServer. KnockKnockClient is based on the EchoClient program in the previous section, Reading from and Writing to a Socket and should be somewhat familiar to you. But we'll go over the program anyway and look at what's happening in the client in the context of what's going on in the server.

When you start the client program, the server should already be running and listening to the port, waiting for a client to request a connection. So, the first thing the client program does is to open a socket that is connected to the server running on the hostname and port specified:

```
kkSocket = new Socket("taranis", 4444);
out = new PrintWriter(kkSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(kkSocket.getInputStream()));
```

When creating its socket, KnockKnockClient uses the host name taranis, the name of a hypothetical machine on our network. When you type in and run this program, change the host name to the name of a machine on your network. This is the machine on which you will run the KnockKnockServer.

The KnockKnockClient program also specifies the port number 4444 when creating its socket. This is a *remote port number*--the number of a port on the server machine--and is the port to which KnockKnockServer is listening. The client's socket is bound to any available *local port*--a port on the client machine. Remember that the server gets a new socket as well. That socket is bound to local port number 4444 on its machine. The server's socket and the client's socket are connected.

Next comes the while loop that implements the communication between the client and the server. The server speaks first, so the client must listen first. The client does this by reading from the input stream attached to the socket. If the server does speak, it says "Bye." and the client exits the loop. Otherwise, the client displays the text to the standard output and then reads the response from the user, who types into the standard input. After the user types a carriage return, the client sends the text to the server through the output stream attached to the socket.

```
while ((fromServer = in.readLine()) != null) {
   System.out.println("Server: " + fromServer);
   if (fromServer.equals("Bye."))
      break;

   fromUser = stdIn.readLine();
   if (fromUser != null) {
      System.out.println("Client: " + fromUser);
      out.println(fromUser);
   }
}
```

The communication ends when the server asks if the client wishes to hear another joke, the client says no, and the server says "Bye."

In the interest of good housekeeping, the client closes its input and output streams and the socket:

```
out.close();
in.close();
```

```
stdIn.close();
kkSocket.close();
```

Running the Programs

You must start the server program first. To do this, run the server program using the Java interpreter, just as you would any other Java application. Remember to run the server on the machine that the client program specifies when it creates the socket.

Next, run the client program. Note that you can run the client on any machine on your network; it does not have to run on the same machine as the server.

If you are too quick, you might start the client before the server has a chance to initialize itself and begin listening on the port. If this happens, you will see a stack trace from the client. If this happens, just restart the client.

If you forget to change the host name in the source code for the KnockKnockClient program, you will see the following error message:

```
Don't know about host: taranis
```

To fix this, modify the KnockKnockClient program and provide a valid host name for your network. Recompile the client program and try again.

If you try to start a second client while the first client is connected to the server, the second client just hangs. The next section, Supporting Multiple Clients, talks about supporting multiple clients.

When you successfully get a connection between the client and server, you will see the following text displayed on your screen:

```
Server: Knock! Knock!
```

Now, you must respond with:

Who's there?

The client echoes what you type and sends the text to the server. The server responds with the first line of one of the many Knock Knock jokes in its repertoire. Now your screen should contain this (the text you typed is in bold):

Server: Knock! Knock!
Who's there?
Client: Who's there?
Server: Turnip

Now, you respond with:

Turnip who?"

Again, the client echoes what you type and sends the text to the server. The server responds with the punch line. Now your screen should contain this:

```
Server: Knock! Knock!
**Who's there?**
Client: Who's there?
Server: Turnip
**Turnip who?**
Client: Turnip who?
Server: Turnip the heat, it's cold in here! Want another? (y/n)
```

If you want to hear another joke, type **y**; if not, type **n**. If you type **y**, the server begins again with "Knock! Knock!" If you type **n**, the server says "Bye." thus causing both the client and the server to exit.

If at any point you make a typing mistake, the KnockKnockServer object catches it and the server responds with a message similar to this:

```
Server: You're supposed to say "Who's there?"!
```

The server then starts the joke over again:

```
Server: Try again. Knock! Knock!
```

Note that the KnockKnockProtocol object is particular about spelling and punctuation but not about capitalization.

Supporting Multiple Clients

To keep the KnockKnockServer example simple, we designed it to listen for and handle a single connection request. However, multiple client requests can come into the same port and, consequently, into the same ServerSocket. Client connection requests are queued at the port, so the server must accept the connections sequentially. However, the server can service them simultaneously through the use of threads - one thread per each client connection.

The basic flow of logic in such a server is this:

```
while (true) {
   accept a connection;
   create a thread to deal with the client;
}
```

The thread reads from and writes to the client connection as necessary.

Try This:

Modify the KnockKnockServer so that it can service multiple clients at the same time. Two classes compose our solution: KKMultiServer andKKMultiServerThread. KKMultiServer loops forever, listening for client connection requests on a ServerSocket. When a request comes in, KKMultiServeraccepts the connection, creates a new KKMultiServerThread object to process it, hands it the socket returned from accept, and starts the thread. Then the server goes back to listening for connection requests. The KKMultiServerThread object communicates to the client by reading from and writing to the socket. Run the new Knock Knock server and then run several clients in succession.

JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

The following simple code fragment gives a simple example of these three steps:

```
public void connectToAndQueryDatabase(String username, String password) {

   Connection con = DriverManager.getConnection(
                    "jdbc:myDriver:myDatabase",
                    username,
                    password);

   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

   while (rs.next()) {
      int x = rs.getInt("a");
      String s = rs.getString("b");
      float f = rs.getFloat("c");
   }
}
```

This short code fragment instantiates a DriverManager object to connect to a database driver and log into the database, instantiates a Statement object that carries your SQL language query to the database; instantiates a ResultSet object that retrieves the results of your query, and executes a simple while loop, which retrieves and displays those results. It's that simple.

Processing SQL Statements with JDBC

In general, to process any SQL statement with JDBC, you follow these steps:

1. Establishing a connection.
2. Create a statement.
3. Execute the query.
4. Process the ResultSet object.
5. Close the connection.

This page uses the following method, CoffeeTables.viewTable, from the tutorial sample to demonstrate these steps. This method outputs the contents of the table COFFEES. This method will be discussed in more detail later in this tutorial:

```
public static void viewTable(Connection con, String dbName)
    throws SQLException {

    Statement stmt = null;
    String query = "select COF_NAME, SUP_ID, PRICE, " +
                   "SALES, TOTAL " +
                   "from " + dbName + ".COFFEES";
    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + "\t" + supplierID +
                               "\t" + price + "\t" + sales +
                               "\t" + total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Establishing Connections

First, establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. This connection is represented by a Connection object. See Establishing a Connection for more information.

Creating Statements

A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.

For example, CoffeesTables.viewTable creates a Statement object with the following code:

```
stmt = con.createStatement();
```

There are three different kinds of statements:

- Statement: Used to implement simple SQL statements with no parameters.
- PreparedStatement: (Extends Statement.) Used for precompiling SQL statements that might contain input parameters. See Using Prepared Statements for more information.
- CallableStatement: (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters. See Stored Procedures for more information.

Executing Queries

To execute a query, call an execute method from Statement such as the following:

- execute: Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResutSet.
- executeQuery: Returns one ResultSet object.
- executeUpdate: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATESQL statements.

For example, CoffeeTables.viewTable executed a Statement object with the following code:

```
ResultSet rs = stmt.executeQuery(query);
```

See Retrieving and Modifying Values from Result Sets for more information.

Processing ResultSet Objects

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in theResultSet object. Initially, the cursor is positioned before the first row. You call various methods defined in the ResultSet object to move the cursor.

For example, CoffeesTables.viewTable repeatedly calls the method ResultSet.next to move the cursor forward by one row. Every time it calls next, the method outputs the data in the row where the cursor is currently positioned:

```
try {
   stmt = con.createStatement();
```

```
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + "\t" + supplierID +
                           "\t" + price + "\t" + sales +
                           "\t" + total);
    }
}
// ...
```

See Retrieving and Modifying Values from Result Sets for more information.

Closing Connections

When you are finished using a Statement, call the method Statement.close to immediately release the resources it is using. When you call this method, its ResultSetobjects are closed.

For example, the method CoffeesTables.viewTable ensures that the Statement object is closed at the end of the method, regardless of any SQLException objects thrown, by wrapping it in a finally block:

```
} finally {
    if (stmt != null) { stmt.close(); }
}
```

JDBC throws an SQLException when it encounters an error during an interaction with a data source. See Handling SQL Exceptions for more information.

In JDBC 4.1, which is available in Java SE release 7 and later, you can use a try-with-resources statement to automatically close Connection, Statement, and ResultSetobjects, regardless of whether an SQLException has been thrown. An automatic resource statement consists of a try statement and one or more declared resources. For example, you can modify CoffeesTables.viewTable so that its Statement object closes automatically, as follows:

```
public static void viewTable(Connection con) throws SQLException {

    String query = "select COF_NAME, SUP_ID, PRICE, " +
                   "SALES, TOTAL " +
                   "from COFFEES";
    try (Statement stmt = con.createStatement()) {

        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + ", " + supplierID +
                               ", " + price + ", " + sales +
                               ", " + total);
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
}
```

The following statement is an try-with-resources statement, which declares one resource, stmt, that will be automatically closed when the try block terminates:

```
try (Statement stmt = con.createStatement()) {
    // ...
}
```

Overview of Prepared Statements

Sometimes it is more convenient to use a PreparedStatement object for sending SQL statements to the database. This special type of statement is derived from the more general class, Statement, that you already know.

If you want to execute a Statement object many times, it usually reduces execution time to use a PreparedStatement object instead.

The main feature of a PreparedStatement object is that, unlike a Statement object, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled. As a result, the PreparedStatement object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first.

Although PreparedStatement objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it. Examples of this are in the following sections.

The following method, CoffeesTable.updateCoffeeSales, stores the number of pounds of coffee sold in the current week in the SALES column for each type of coffee, and updates the total number of pounds of coffee sold in the TOTAL column for each type of coffee:

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)
   throws SQLException {

   PreparedStatement updateSales = null;
   PreparedStatement updateTotal = null;

   String updateString =
      "update " + dbName + ".COFFEES " +
      "set SALES = ? where COF_NAME = ?";

   String updateStatement =
      "update " + dbName + ".COFFEES " +
      "set TOTAL = TOTAL + ? " +
      "where COF_NAME = ?";

   try {
      con.setAutoCommit(false);
      updateSales = con.prepareStatement(updateString);
      updateTotal = con.prepareStatement(updateStatement);

      for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
         updateSales.setInt(1, e.getValue().intValue());
         updateSales.setString(2, e.getKey());
         updateSales.executeUpdate();
         updateTotal.setInt(1, e.getValue().intValue());
         updateTotal.setString(2, e.getKey());
         updateTotal.executeUpdate();
         con.commit();
      }
   } catch (SQLException e ) {
      JDBCTutorialUtilities.printSQLException(e);
      if (con != null) {
         try {
            System.err.print("Transaction is being rolled back");
            con.rollback();
        } catch(SQLException excep) {
            JDBCTutorialUtilities.printSQLException(excep);
        }
    }
  } finally {
    if (updateSales != null) {
        updateSales.close();
    }
    if (updateTotal != null) {
        updateTotal.close();
    }
    con.setAutoCommit(true);
  }
}
```

Creating a PreparedStatement Object

The following creates a PreparedStatement object that takes two input parameters:

```
String updateString =
  "update " + dbName + ".COFFEES " +
  "set SALES = ? where COF_NAME = ?";
updateSales = con.prepareStatement(updateString);
```

Supplying Values for PreparedStatement Parameters

You must supply values in place of the question mark placeholders (if there are any) before you can execute a PreparedStatement object. Do this by calling one of the setter methods defined in the PreparedStatement class. The following statements supply the two question mark placeholders in the PreparedStatement named updateSales:

```
updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());
```

The first argument for each of these setter methods specifies the question mark placeholder. In this example, setInt specifies the first placeholder and setString specifies the second placeholder.

After a parameter has been set with a value, it retains that value until it is reset to another value, or the method clearParameters is called. Using the PreparedStatementobject updateSales, the following code fragment illustrates reusing a prepared statement after resetting the value of one of its parameters and leaving the other one the same:

```
// changes SALES column of French Roast
//row to 100

updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();

// changes SALES column of Espresso row to 100
// (the first parameter stayed 100, and the second
// parameter was reset to "Espresso")

updateSales.setString(2, "Espresso");
updateSales.executeUpdate();
```

Using Loops to Set Values

You can often make coding easier by using a for loop or a while loop to set values for input parameters.

The CoffeesTable.updateCoffeeSales method uses a for-each loop to repeatedly set values in the PreparedStatement objects updateSales and updateTotal:

```
for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {

  updateSales.setInt(1, e.getValue().intValue());
  updateSales.setString(2, e.getKey());

  // ...
}
```

The method CoffeesTable.updateCoffeeSales takes one argument, HashMap. Each element in the HashMap argument contains the name of one type of coffee and the number of pounds of that type of coffee sold during the current week. The for-each loop iterates through each element of the HashMap argument and sets the appropriate question mark placeholders in updateSales and updateTotal.

Executing PreparedStatement Objects

As with Statement objects, to execute a PreparedStatement object, call an execute statement: executeQuery if the query returns only one ResultSet (such as a SELECTSQL statement), executeUpdate if the query does not return a ResultSet (such as an UPDATE SQL statement), or execute if the query might return more than oneResultSet object. Both PreparedStatement objects in CoffeesTable.updateCoffeeSales contain UPDATE SQL statements, so both are executed by callingexecuteUpdate:

```
updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());
```
updateSales.executeUpdate();

```
updateTotal.setInt(1, e.getValue().intValue());
updateTotal.setString(2, e.getKey());
updateTotal.executeUpdate();
con.commit();
```

No arguments are supplied to executeUpdate when they are used to execute updateSales and updateTotals; both PreparedStatement objects already contain the SQL statement to be executed.

Note: At the beginning of CoffeesTable.updateCoffeeSales, the auto-commit mode is set to false:

```
con.setAutoCommit(false);
```

Consequently, no SQL statements are committed until the method commit is called. For more information about the auto-commit mode, see Transactions.

Return Values for the executeUpdate Method

Whereas executeQuery returns a ResultSet object containing the results of the query sent to the DBMS, the return value for executeUpdate is an int value that indicates how many rows of a table were updated. For instance, the following code shows the return value of executeUpdate being assigned to the variable n:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
```
// n = 1 because one row had a change in it

The table COFFEES is updated; the value 50 replaces the value in the column SALES in the row for Espresso. That update affects one row in the table, so n is equal to 1.

When the method executeUpdate is used to execute a DDL (data definition language) statement, such as in creating a table, it returns the int value of 0. Consequently, in the following code fragment, which executes the DDL statement used to create the table COFFEES, n is assigned a value of 0:

// n = 0
```
int n = executeUpdate(createTableCoffees);
```

Note that when the return value for executeUpdate is 0, it can mean one of two things:

- The statement executed was an update statement that affected zero rows.
- The statement executed was a DDL statement.

Using Transactions

There are times when you do not want one statement to take effect unless another one completes. For example, when the proprietor of The Coffee Break updates the amount of coffee sold each week, the proprietor will also want to update the total amount sold to date. However, the amount sold per week and the total amount sold should be updated at the same time; otherwise, the data will be inconsistent. The way to be sure that either both actions occur or neither action occurs is to use a transaction. A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.

Disabling Auto-Commit Mode

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. (To be more precise, the default is for a SQL statement to be committed when it is completed, not when it is executed. A statement is completed when all of its result sets and update counts have been retrieved. In almost all cases, however, a statement is completed, and therefore committed, right after it is executed.)

The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. This is demonstrated in the following code, where con is an active connection:

```
con.setAutoCommit(false);
```

Committing Transactions

After the auto-commit mode is disabled, no SQL statements are committed until you call the method commit explicitly. All statements executed after the previous call to the method commit are included in the current transaction and committed together as a unit. The following method, CoffeesTable.updateCoffeeSales, in which con is an active connection, illustrates a transaction:

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)
   throws SQLException {

   PreparedStatement updateSales = null;
   PreparedStatement updateTotal = null;

   String updateString =
            "update " + dbName + ".COFFEES " +
            "set SALES = ? where COF_NAME = ?";

        String updateStatement =
            "update " + dbName + ".COFFEES " +
            "set TOTAL = TOTAL + ? " +
            "where COF_NAME = ?";

        try {
            con.setAutoCommit(false);
            updateSales = con.prepareStatement(updateString);
            updateTotal = con.prepareStatement(updateStatement);

            for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
                updateSales.setInt(1, e.getValue().intValue());
                updateSales.setString(2, e.getKey());
                updateSales.executeUpdate();
                updateTotal.setInt(1, e.getValue().intValue());
                updateTotal.setString(2, e.getKey());
                updateTotal.executeUpdate();
                con.commit();
            }
        } catch (SQLException e ) {
            JDBCTutorialUtilities.printSQLException(e);
            if (con != null) {
                try {
                    System.err.print("Transaction is being rolled back");
                    con.rollback();
                } catch(SQLException excep) {
                    JDBCTutorialUtilities.printSQLException(excep);
                }
            }
        } finally {
            if (updateSales != null) {
                updateSales.close();
            }
            if (updateTotal != null) {
                updateTotal.close();
            }
            con.setAutoCommit(true);
        }
    }
```

In this method, the auto-commit mode is disabled for the connection con, which means that the two prepared statements updateSales and updateTotal are committed together when the method commit is called. Whenever the commit method is called (either automatically when auto-commit mode is enabled or explicitly when it is disabled), all changes resulting from statements in the transaction are made permanent. In this case, that means that the SALES and TOTAL columns for Colombian coffee have been changed to 50 (if TOTAL had been 0 previously) and will retain this value until they are changed with another update statement.

The statement con.setAutoCommit(true); enables auto-commit mode, which means that each statement is once again committed automatically when it is completed. Then, you are back to the default state where you do not have to call the method commit yourself. It is advisable to disable the auto-commit mode only during the transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users.

JDK 5.0 new features

Enum Types

An *enum type* is a type whose *fields* consist of a fixed set of constants. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Because they are constants, the names of an enum type's fields are in uppercase letters.

In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
   THURSDAY, FRIDAY, SATURDAY
}
```

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

Annotations

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compiler-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at runtime.

Annotations can be applied to a program's declarations of classes, fields, methods, and other program elements.

The annotation appears first, often (by convention) on its own line, and may include *elements* with named or unnamed values:

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass() { }
```

or

```
@SuppressWarnings(value = "unchecked")
void myMethod() { }
```

If there is just one element named "value," then the name may be omitted, as in:

```
@SuppressWarnings("unchecked")
void myMethod() { }
```

Also, if an annotation has no elements, the parentheses may be omitted, as in:

```
@Override
void mySuperMethod() { }
```

Annotations Used by the Compiler

There are three annotation types that are predefined by the language specification itself: @Deprecated, @Override, and @SuppressWarnings.

@Deprecated—the @Deprecated annotation indicates that the marked element is *deprecated* and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. The use of the "@" symbol in both Javadoc comments and in annotations is not coincidental — they are related conceptually. Also, note that the Javadoc tag starts with a lowercase "d" and the annotation starts with an uppercase "D".

```
   // Javadoc comment follows
    /**
     * @deprecated
     * explanation of why it
     * was deprecated
     */
    @Deprecated
    static void deprecatedMethod() { }
}
```

@Override—the @Override annotation informs the compiler that the element is meant to override an element declared in a superclass (overriding methods will be discussed in the the lesson titled "Interfaces and Inheritance").

```
// mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

While it's not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with @Override fails to correctly override a method in one of its superclasses, the compiler generates an error.

@SuppressWarnings—the @SuppressWarnings annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the example below, a deprecated method is used and the compiler would normally generate a warning. In this case, however, the annotation causes the warning to be suppressed.

```
// use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation")
 void useDeprecatedMethod() {
    // deprecation warning
    // - suppressed
    objectOne.deprecatedMethod();
 }
```

Every compiler warning belongs to a category. The Java Language Specification lists two categories: "deprecation" and "unchecked." The "unchecked" warning can occur when interfacing with legacy code written before the advent of generics (discussed in the lesson titled "Generics"). To suppress more than one category of warnings, use the following syntax:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

The Reflection API

Uses of Reflection

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

Classes

Every object is either a reference or primitive type. Reference types all inherit from java.lang.Object. Classes, enums, arrays, and interfaces are all reference types. There is a fixed set of primitive types: boolean, byte, short, int, long, char, float, and double. Examples of reference types include java.lang.String, all of the wrapper classes for primitive types such as java.lang.Double, the interface java.io.Serializable, and the enum javax.swing.SortOrder.

For every type of object, the Java virtual machine instantiates an immutable instance of java.lang.Class which provides methods to examine the runtime properties of the object including its members and type information. Class also provides the ability to create new classes and objects. Most importantly, it is the entry point for all of the Reflection APIs.

Retrieving Class Objects

The entry point for all reflection operations is java.lang.Class. With the exception of java.lang.reflect.ReflectPermission, none of the classes injava.lang.reflect have public constructors. To get to these classes, it is necessary to invoke appropriate methods on Class. There are several ways to get a Classdepending on whether the code has access to an object, the name of class, a type, or an existing Class.

Object.getClass()

If an instance of an object is available, then the simplest way to get its Class is to invoke Object.getClass(). Of course, this only works for reference types which all inherit from Object. Some examples follow.

```
Class c = "foo".getClass();
```

Returns the Class for String

```
Class c = System.console().getClass();
```

There is a unique console associated with the virtual machine which is returned by the static method System.console(). The value returned by getClass() is the Classcorresponding to java.io.Console.

```
enum E { A, B }
Class c = A.getClass();
```

A is is an instance of the enum E; thus getClass() returns the Class corresponding to the enumeration type E.

```
byte[] bytes = new byte[1024];
```

```
Class c = bytes.getClass();
```

Since arrays are Objects, it is also possible to invoke getClass() on an instance of an array. The returned Class corresponds to an array with component type byte.

```
import java.util.HashSet;
import java.util.Set;

Set<String> s = new HashSet<String>();
Class c = s.getClass();
```

In this case, java.util.Set is an interface to an object of type java.util.HashSet. The value returned by getClass() is the class corresponding tojava.util.HashSet.

Fields

A *field* is a class, interface, or enum with an associated value. Methods in the java.lang.reflect.Field class can retrieve information about the field, such as its name, type, modifiers, and annotations. (The section Examining Class Modifiers and Types in the Classes lesson describes how to retrieve annotations.) There are also methods which enable dynamic access and modification of the value of the field.

Obtaining Field Types

A field may be either of primitive or reference type. There are eight primitive types: boolean, byte, short, int, long, char, float, and double. A reference type is anything that is a direct or indirect subclass of java.lang.Object including interfaces, arrays, and enumerated types.

The FieldSpy example prints the field's type and generic type given a fully-qualified binary class name and field name.

```
import java.lang.reflect.Field;
import java.util.List;

public class FieldSpy<T> {
   public boolean[][] b = {{ false, false }, { true, true } };
   public String name  = "Alice";
   public List<Integer> list;
   public T val;

   public static void main(String... args) {
```

```
            try {
                Class<?> c = Class.forName(args[0]);
                Field f = c.getField(args[1]);
                System.out.format("Type: %s%n", f.getType());
                System.out.format("GenericType: %s%n", f.getGenericType());

        // production code should handle these exceptions more gracefully
            } catch (ClassNotFoundException x) {
                x.printStackTrace();
            } catch (NoSuchFieldException x) {
                x.printStackTrace();
            }
    }
}
```

Sample output to retrieve the type of the three public fields in this class (b, name, and the parameterized type list), follows. User input is in italics.

```
$ java FieldSpy FieldSpy b
Type: class [[Z
GenericType: class [[Z
$ java FieldSpy FieldSpy name
Type: class java.lang.String
GenericType: class java.lang.String
$ java FieldSpy FieldSpy list
Type: interface java.util.List
GenericType: java.util.List<java.lang.Integer>
$ java FieldSpy FieldSpy val
Type: class java.lang.Object
GenericType: T
```

The type for the field b is two-dimensional array of boolean. The syntax for the type name is described in Class.getName().

The type for the field val is reported as java.lang.Object because generics are implemented via *type erasure* which removes all information regarding generic types during compilation. Thus T is replaced by the upper bound of the type variable, in this case, java.lang.Object.

Field.getGenericType() will consult the Signature Attribute in the class file if it's present. If the attribute isn't available, it falls back on Field.getType() which was not changed by the introduction of generics. The other methods in reflection with name getGeneric*Foo* for some value of *Foo* are implemented similarly.

Obtaining Method Type Information

A method declaration includes the name, modifiers, parameters, return type, and list of throwable exceptions. The java.lang.reflect.Method class provides a way to obtain this information.

The MethodSpy example illustrates how to enumerate all of the declared methods in a given class and retrieve the return, parameter, and exception types for all the methods of the given name.

```
import java.lang.reflect.Method;
import java.lang.reflect.Type;
import static java.lang.System.out;

public class MethodSpy {
   private static final String  fmt = "%24s: %s%n";

   // for the morbidly curious
   <E extends RuntimeException> void genericThrow() throws E {}

   public static void main(String... args) {
        try {
            Class<?> c = Class.forName(args[0]);
            Method[] allMethods = c.getDeclaredMethods();
            for (Method m : allMethods) {
                if (!m.getName().equals(args[1])) {
                    continue;
                }
                out.format("%s%n", m.toGenericString());

                out.format(fmt, "ReturnType", m.getReturnType());
                out.format(fmt, "GenericReturnType", m.getGenericReturnType());

                Class<?>[] pType  = m.getParameterTypes();
                Type[] gpType = m.getGenericParameterTypes();
                for (int i = 0; i < pType.length; i++) {
                    out.format(fmt,"ParameterType", pType[i]);
                    out.format(fmt,"GenericParameterType", gpType[i]);
                }

                Class<?>[] xType  = m.getExceptionTypes();
                Type[] gxType = m.getGenericExceptionTypes();
                for (int i = 0; i < xType.length; i++) {
                        out.format(fmt,"ExceptionType", xType[i]);
                        out.format(fmt,"GenericExceptionType", gxType[i]);
                    }
            }

        // production code should handle these exceptions more gracefully
        } catch (ClassNotFoundException x) {
            x.printStackTrace();
        }
    }
}
```

Here is the output for Class.getConstructor() which is an example of a method with parameterized types and a variable number of parameters.

```
$ java MethodSpy java.lang.Class getConstructor
public java.lang.reflect.Constructor<T> java.lang.Class.getConstructor
  (java.lang.Class<?>[]) throws java.lang.NoSuchMethodException,
  java.lang.SecurityException
              ReturnType: class java.lang.reflect.Constructor
       GenericReturnType: java.lang.reflect.Constructor<T>
           ParameterType: class [Ljava.lang.Class;
    GenericParameterType: java.lang.Class<?>[]
           ExceptionType: class java.lang.NoSuchMethodException
    GenericExceptionType: class java.lang.NoSuchMethodException
           ExceptionType: class java.lang.SecurityException
    GenericExceptionType: class java.lang.SecurityException
```

This is the actual declaration of the method in source code:

```
public Constructor<T> getConstructor(Class<?>... parameterTypes)
```

First note that the return and parameter types are generic. Method.getGenericReturnType() will consult the Signature Attribute in the class file if it's present. If the attribute isn't available, it falls back on Method.getReturnType() which was not changed by the introduction of generics. The other methods with name getGeneric*Foo*() for some value of *Foo* in reflection are implemented similarly.

Next, notice that the last (and only) parameter, parameterType, is of variable arity (has a variable number of parameters) of type java.lang.Class. It is represented as a single-dimension array of type java.lang.Class. This can be distinguished from a parameter that is explicitly an array of java.lang.Class by invokingMethod.isVarArgs(). The syntax for the returned values of Method.get*Types() is described in Class.getName().

Arrays

Arrays have a component type and a length (which is not part of the type). Arrays may be maniuplated either in their entirety or component by component. Reflection provides the java.lang.reflect.Array class for the latter purpose.

Identifying Array Types

Array types may be identified by invoking Class.isArray(). To obtain a Class use one of the methods described in Retrieving Class Objects section of this trail.

The ArrayFind example identifies the fields in the named class that are of array type and reports the component type for each of them.

```
import java.lang.reflect.Field;
import java.lang.reflect.Type;
import static java.lang.System.out;

public class ArrayFind {
  public static void main(String... args) {
        boolean found = false;
        try {
          Class<?> cls = Class.forName(args[0]);
          Field[] flds = cls.getDeclaredFields();
          for (Field f : flds) {
                Class<?> c = f.getType();
                if (c.isArray()) {
                  found = true;
                  out.format("%s%n"
                + "           Field: %s%n"
                            + "           Type: %s%n"
                            + "  Component Type: %s%n",
                            f, f.getName(), c, c.getComponentType());
                }
          }
          if (!found) {
                out.format("No array fields%n");
          }

    // production code should handle this exception more gracefully
        } catch (ClassNotFoundException x) {
          x.printStackTrace();
        }
    }
}
```

The syntax for the returned value of Class.get*Type() is described in Class.getName(). The number of '[' characters at the beginning of the type name indicates the number of dimensions (i.e. depth of nesting) of the array.

Samples of the output follows. User input is in italics. An array of primitive type byte:

```
$java ArrayFind java.nio.ByteBuffer
final byte[] java.nio.ByteBuffer.hb
          Field: hb
           Type: class [B
  Component Type: byte
```

Generics

JDK 5.0 introduces several new extensions to the Java programming language. One of these is the introduction of *generics*.

This trail is an introduction to generics. You may be familiar with similar constructs from other languages, most notably C++ templates. If so, you'll see that there are both similarities and important differences. If you are unfamiliar with look-a-alike constructs from elsewhere, all the better; you can start fresh, without having to unlearn any misconceptions.

Generics allow you to abstract over types. The most common examples are container types, such as those in the Collections hierarchy.

Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required.

Of course, the cast not only introduces clutter. It also introduces the possibility of a run time error, since the programmer may be mistaken.

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer>
   myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); // 2'
Integer x = myIntList.iterator().next(); // 3'
```

Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>. We say that Listis a generic interface that takes a *type parameter*--in this case, Integer. We also specify a type parameter when creating the list object.

Note, too, that the cast on line 3' is gone.

Defining Simple Generics

Here is a small excerpt from the definitions of the interfaces List and Iterator in package java.util:

```
public interface List <E> {
   void add(E x);
   Iterator<E> iterator();
}

public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

This code should all be familiar, except for the stuff in angle brackets. Those are the declarations of the *formal type parameters* of the interfaces List and Iterator.

Generics and Subtyping

Let's test your understanding of generics. Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); // 1
List<Object> lo = ls; // 2
```

Line 1 is certainly legal. The trickier part of the question is line 2. This boils down to the question: is a List of String a List of Object. Most people instinctively answer, "Sure!"

Well, take a look at the next few lines:

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4: Attempts to assign an Object to a String!
```

Here we've aliased ls and lo. Accessing ls, a list of String, through the alias lo, we can insert arbitrary objects into it. As a result ls does not hold just Strings anymore, and when we try and get something out of it, we get a rude surprise.

The Java compiler will prevent this from happening of course. Line 2 will cause a compile time error.

In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that G<Foo> is a subtype of G<Bar>. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.

We should not assume that collections don't change. Our instinct may lead us to think of these things as immutable.

Wildcards

Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):

```
void printCollection(Collection c) {
   Iterator i = c.iterator();
   for (k = 0; k < c.size(); k++) {
      System.out.println(i.next());
   }
}
```

And here is a naive attempt at writing it using generics (and the new for loop syntax):

```
void printCollection(Collection<Object> c) {
   for (Object e : c) {
      System.out.println(e);
   }
}
```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes Collection<Object>, which, as we've just demonstrated, is **not** a supertype of all kinds of collections!

So what **is** the supertype of all kinds of collections? It's written Collection<?> (pronounced "collection of unknown"), that is, a collection whose element type matches anything. It's called a **wildcard type** for obvious reasons. We can write:

```
void printCollection(Collection<?> c) {
   for (Object e : c) {
      System.out.println(e);
   }
}
```

and now, we can call it with any type of collection. Notice that inside printCollection(), we can still read elements from c and give them type Object. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // Compile time error
```

Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.

On the other hand, given a List<?>, we **can** call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected.

Programming with Assertions

- An *assertion* is a statement in the Java^{TM} programming language that enables you to test your assumptions about your program. For example, if you write a method that calculates the speed of a particle, you might assert that the calculated speed is less than the speed of light.
- Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error. By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.
- Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs. As an added benefit, assertions serve to document the inner workings of your program, enhancing maintainability.

Introduction

The assertion statement has two forms. The first, simpler form is:

assert *Expression*$_1$;

where *Expression*$_1$ is a boolean expression. When the system runs the assertion, it evaluates *Expression*$_1$ and if it is false throws an AssertionError with no detail message.

The second form of the assertion statement is:

assert *Expression*$_1$: *Expression*$_2$;

where:

- *Expression*$_1$ is a boolean expression.
- *Expression*$_2$ is an expression that has a value. (It cannot be an invocation of a method that is declared void.)

Use this version of the assert statement to provide a detail message for the AssertionError. The system passes the value of *Expression*$_2$ to the appropriate AssertionError constructor, which uses the string representation of the value as the error's detail message.

The purpose of the detail message is to capture and communicate the details of the assertion failure. The message should allow you to diagnose and ultimately fix the error that led the assertion to fail. Note that the detail message is *not* a user-level error message, so it is generally unnecessary to make these messages understandable in isolation, or to internationalize them. The detail message is meant to be interpreted in the context of a full stack trace, in conjunction with the source code containing the failed assertion.

Like all uncaught exceptions, assertion failures are generally labeled in the stack trace with the file and line number from which they were thrown. The second form of the assertion statement should be used in preference to the first only when the program has some additional information that might help diagnose the failure. For example, if *Expression$_1$* involves the relationship between two variables x and y, the second form should be used. Under these circumstances, a reasonable candidate for *Expression$_2$* would be "x: " + x + ", y: " + y.

In some cases *Expression$_1$* may be expensive to evaluate. For example, suppose you write a method to find the minimum element in an unsorted list, and you add an assertion to verify that the selected element is indeed the minimum. The work done by the assert will be at least as expensive as the work done by the method itself. To ensure that assertions are not a performance liability in deployed applications, assertions can be enabled or disabled when the program is started, and are disabled by default. Disabling assertions eliminates their performance penalty entirely. Once disabled, they are essentially equivalent to *empty statements* in semantics and performance.

Putting Assertions Into Your Code

There are many situations where it is good to use assertions.

There are also a few situations where you should *not* use them:

- Do *not* use assertions for argument checking in public methods.

  Argument checking is typically part of the published specifications (or *contract*) of a method, and these specifications must be obeyed whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, IndexOutOfBoundsException, orNullPointerException). An assertion failure will not throw an appropriate exception.

- Do *not* use assertions to do any work that your application requires for correct operation.

  Because assertions may be disabled, programs must not assume that the boolean expression contained in an assertion will be evaluated. Violating this rule has dire consequences. For example, suppose you wanted to remove all of the null elements from a list names, and knew that the list contained one or more nulls. It would be wrong to do this:

  ```
  // Broken! - action is contained in assertion
  assert names.remove(null);
  ```

  The program would work fine when asserts were enabled, but would fail when they were disabled, as it would no longer remove the null elements from the list. The correct idiom is to perform the action before the assertion and then assert that the action succeeded:

  ```
  // Fixed - action precedes assertion
  boolean nullsRemoved = names.remove(null);
  ```

```
assert nullsRemoved;  // Runs whether or not asserts are enabled
```

As a rule, the expressions contained in assertions should be free of *side effects*: evaluating the expression should not affect any state that is visible after the evaluation is complete. One exception to this rule is that assertions can modify state that is used only from within other assertions.

Compiling Files That Use Assertions

In order for the javac compiler to accept code containing assertions, you must use the -source 1.4 command-line option as in this example:

```
javac -source 1.4 MyClass.java
```

Enabling and Disabling Assertions

By default, assertions are disabled at runtime. Two command-line switches allow you to selectively enable or disable assertions.

To enable assertions at various granularities, use the -enableassertions, or -ea, switch. To disable assertions at various granularities, use the -disableassertions, or -da, switch. You specify the granularity with the arguments that you provide to the switch:

- no arguments
  Enables or disables assertions in all classes except system classes.
- *packageName*...
  Enables or disables assertions in the named package and any subpackages.
- ...
  Enables or disables assertions in the unnamed package in the current working directory.
- *className*
  Enables or disables assertions in the named class

For example, the following command runs a program, BatTutor, with assertions enabled in only package com.wombat.fruitbat and its subpackages:

```
java -ea:com.wombat.fruitbat... BatTutor
```

If a single command line contains multiple instances of these switches, they are processed in order before loading any classes. For example, the following command runs the BatTutor program with assertions enabled in package com.wombat.fruitbat but disabled in class com.wombat.fruitbat.Brickbat:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat BatTutor
```

The above switches apply to all class loaders. With one exception, they also apply to *system classes* (which do not have an explicit class loader). The exception concerns the switches with no arguments, which (as indicated above) do not apply to system classes. This behavior makes it easy to enable asserts in all classes except for system classes, which is commonly desirable.

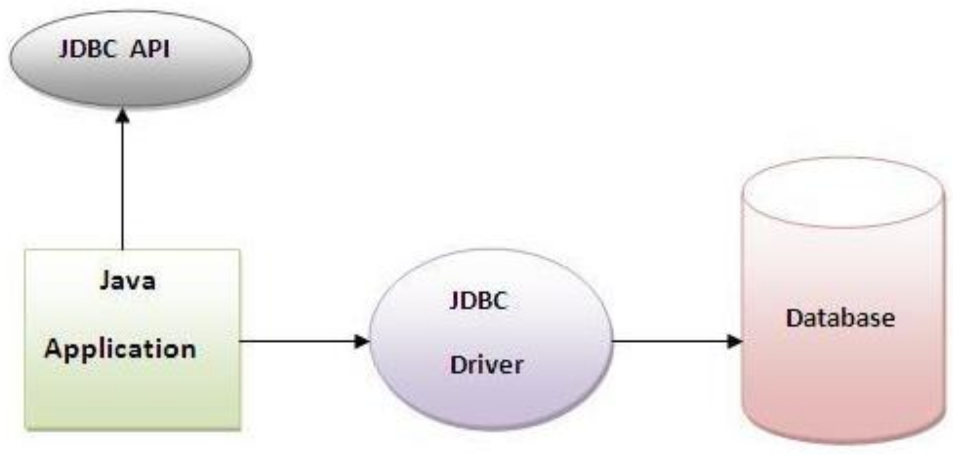
Before assertions were available, many programmers used comments to indicate their assumptions concerning a program's behavior. For example, you might have written something like this to explain your assumption about an else clause in a multiway if-statement:

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // We know (i % 3 == 2)
    ...
}
```

You should now **use an assertion whenever you would have written a comment that asserts an invariant**. For example, you should rewrite the previous if-statement like this:

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

JDBC Tutorials
Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



Why use JDBC
Before JDBC, ODBC API was the database API to connect and execute query with the database. But ODBC API uses ODBC driver which is written in C Language (ie platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java Language).

What is API

API (Application Programming Interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follows to communicate with each other. An API can be created for application, libraries and operating system.

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers.

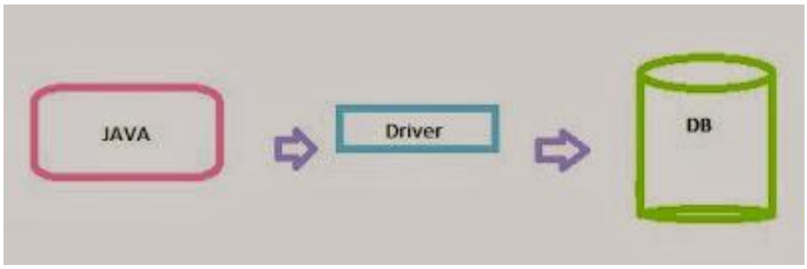
1. JDBC-ODBC bridge driver
2. Native API driver (partially java driver)
3. Network protocol driver (fully java driver)
4. Thin driver (fully java driver)

Jdbc-Driver Types With Examples In Java

Before Discussing in detail about the type of Jdbc-driver . We need to first understand the meaning of term Driver .
So , What is Driver ?

The concept is that a Driver should be a one point contact for all interactions between your Java App. and the DB.

This is the core concept of JDBC



Read Also : JDBC Interview Questions

JDBC (Java Database Connectivity) a specification pitched in by the JCP team (from Java), which gives a contract level agreement between the JAVA and the Database

The specifications declared as part of JDBC are available as part of java.sql package.

Types of JDBC Drivers

The various types of JDBC Drivers are based on the **WAY** the above contract level agreement (shown in the image) is **IMPLEMENTED** by various coders.

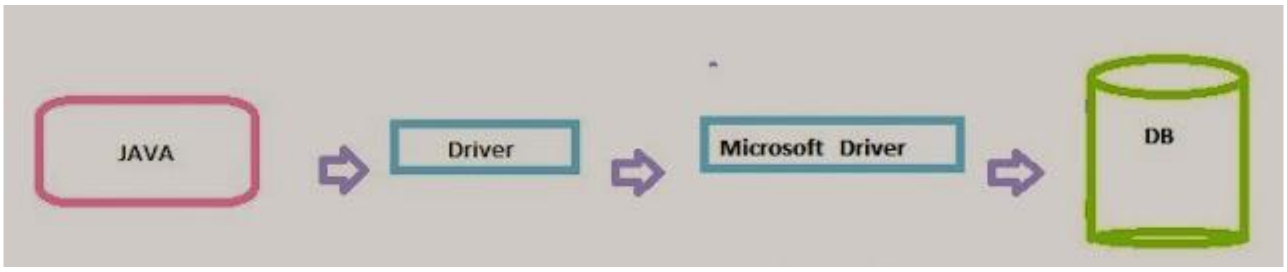
That means , the JDBC specification given by the JCP team only defines the various interfaces, termed as JDBC API . It is up to the coders/developers/implementers who can implement those interfaces, in their own ways.

Based on the ways followed, we can classify them into four types.

JDBC Driver – Type 1 (JDBC ODBC Bridge)

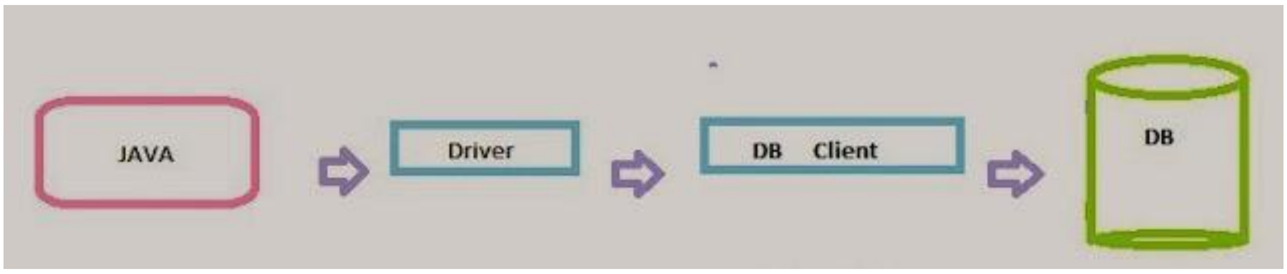
This is an approach wherein the implemented class in Java makes calls to the code written in Microsoft languages (native), which speaks directly to the database.

The first category of JDBC drivers provides a bridge between the JDBC and the ODBC API . The bridge translates the standard JDBC calls and sends them to the ODBC data source via ODBC libraries .



JDBC Driver – Type 2 (Part Native Driver)

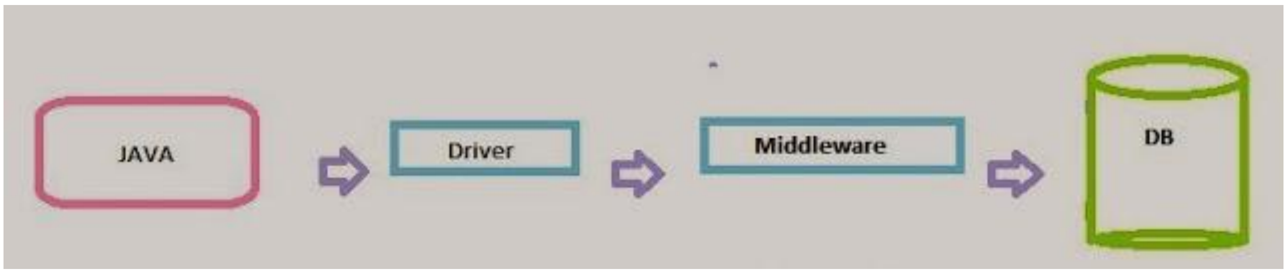
This is an approach wherein the implemented class in Java makes calls to the code written from the database provider (native), which speaks directly to the database.



JDBC Driver – Type 3

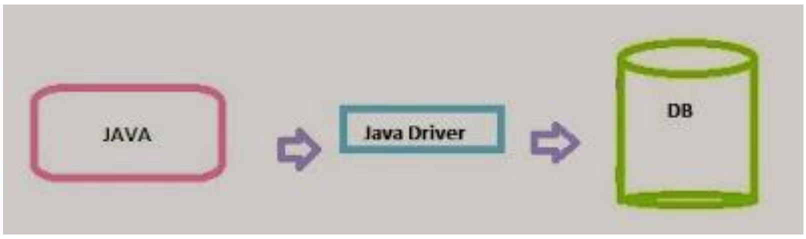
This is an approach wherein the implemented class in Java makes calls to the code written from application server providers, which speaks directly to the database. More exploration on the way the Java Driver interacts with the Middleware is required here.

The Java client application sends a JDBC calls through a JDBC driver to the intermediate data access server ,which completes the request to the data source using another driver . This driver uses a database independent protocol , to communicate database request to a server component which then translate the request into a DB specific protocol .



JDBC Driver – Type 4 (Thin Driver)

This is an approach wherein the implemented class in Java (implemented by the database provider) speaks directly to the database.
In other words , it is a pure Java library that translates JDBC request directly to a Database specific protocol .



JDBC - ODBC bridge driver

The JDBC ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver convert JDBC method calls into the ODBC function call. This is now discouraged because of thin driver.

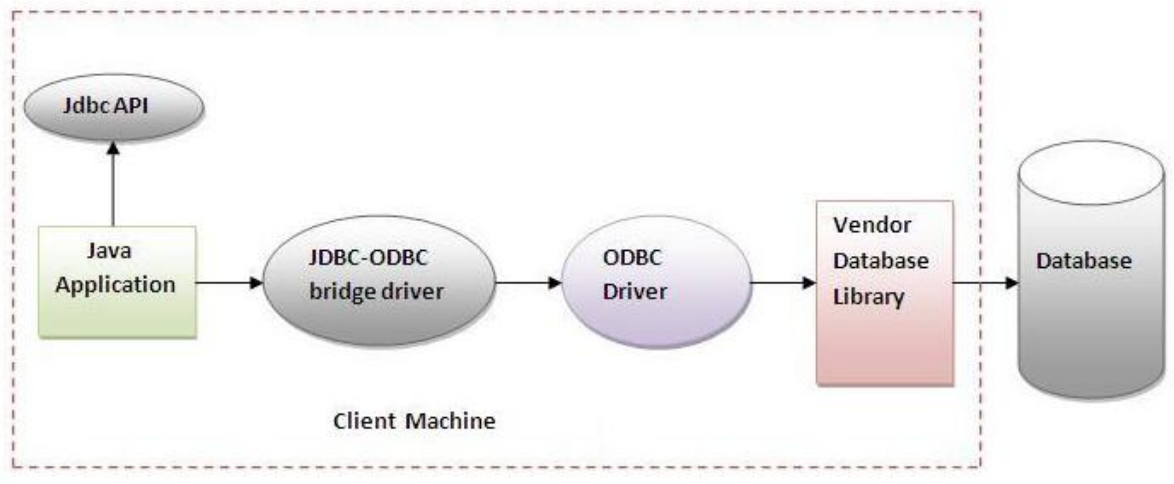


Figure- JDBC-ODBC Bridge Driver

Advantage
1. Easy to use

2. Can be easily connected to any database.

Disadvantage

1. Performance degraded because JDBC method call is converted into the ODBC function calls.
2. The ODBC driver needs to be installed on the client machine.

Native API Driver

The native API driver uses the client-side libraries of the database. The driver convert JDBC methods calls into native calls of the database API. It is not written entirely in Java.

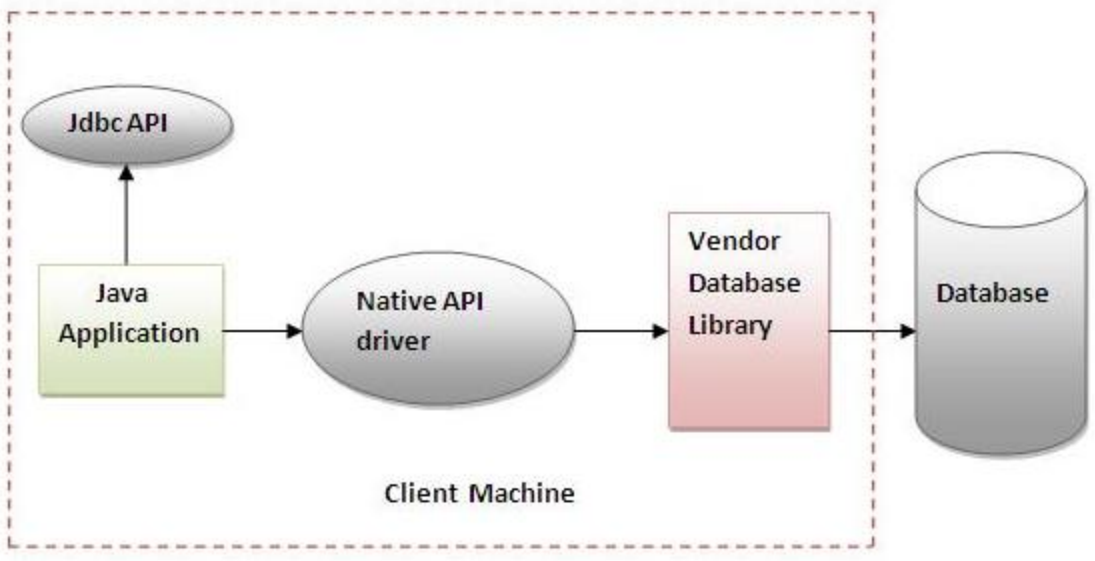


Figure- Native API Driver

Advantage

1. Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage

1. The native driver needs to be installed on the each client machine.
2. The vendor client library need to be installed on client machine.

Network Protocol Driver

The network protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in Java.

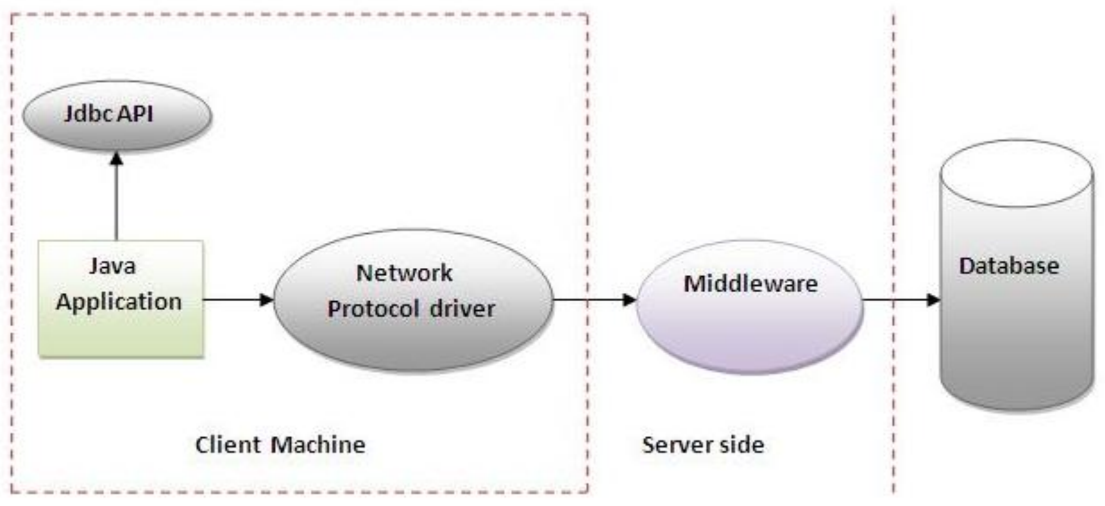


Figure- Network Protocol Driver

Advantage

1. No client side library is required because of application server that can perform many task like auditing, load balancing, logging etc.

Disadvantage

1. Network support is required on client machine.
2. Requires database-specific coding to done in the middle tier.
3. Maintenance of network protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Thin Driver

The thin driver convert JDBC calls directly into the vendor - specific protocol. That is why it is known as thin driver. It is fully written in Java language.

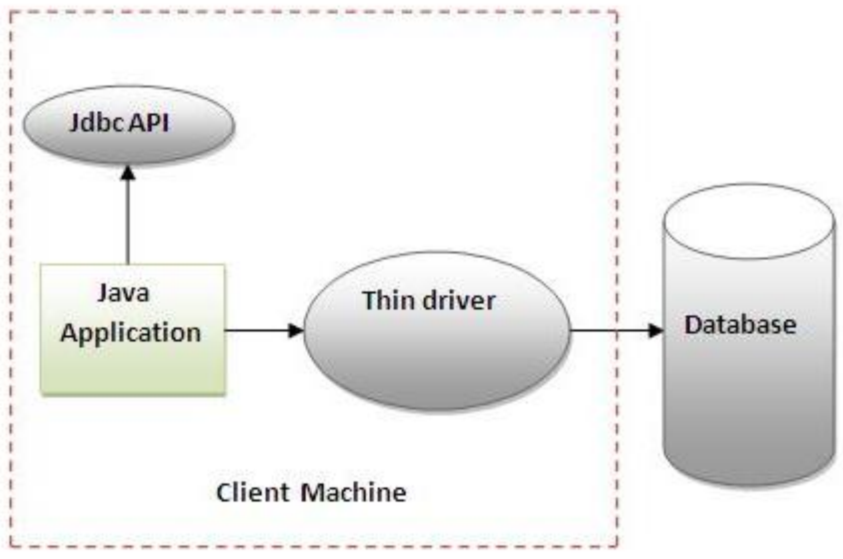


Figure- Thin Driver

Advantage

1. Better performance that all other drivers.
2. No software is required at client side or server side.

Disadvantage

1. Drivers depends on the database.

Steps to connect the Database

1. Load the Driver :

Driver is a piece of software. These driver are provided by the vendor we have to load it manually which is used to connect the database. To load the Driver

First approach

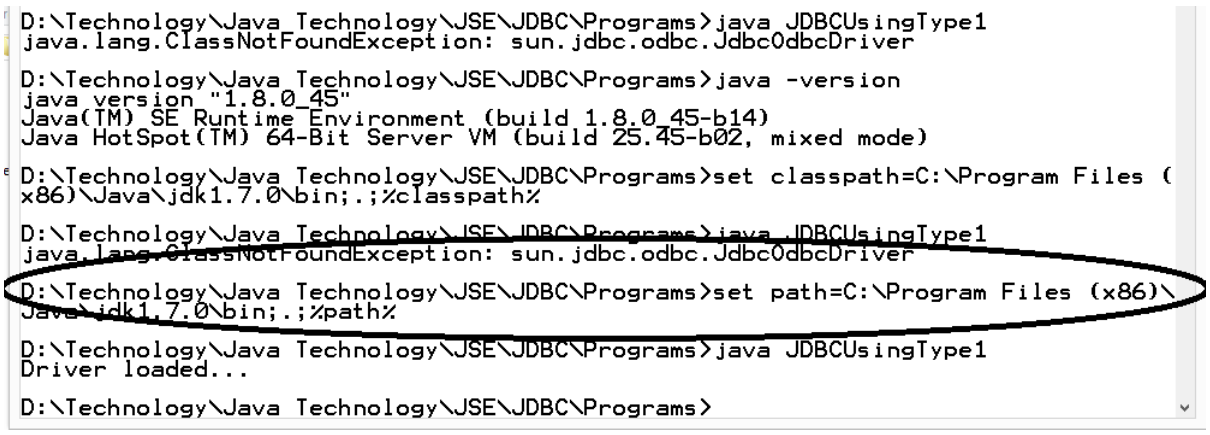
Class.forName("Driver Name"); Here the type 1 driver name is sun.jdbc.odbc.JdbcOdbcDriver.

Second approach

Driver dr = new sun.jdbc.odbc.JdbcOdbcDriver; Here created the instance of the Driver interface reference.

DriverManager.registerDriver(dr);

If you try to load the type1 driver in jdk 1.8 it is removed so make sure you are using jdk less than version 1.8 or set the path to the jdk 1.7



```
D:\Technology\Java_Technology\JSE\JDBC\Programs>java JDBCUsingType1
java.lang.ClassNotFoundException: sun.jdbc.odbc.JdbcOdbcDriver

D:\Technology\Java_Technology\JSE\JDBC\Programs>java -version
java version "1.8.0_45"
Java(TM) SE Runtime Environment (build 1.8.0_45-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode)

D:\Technology\Java Technology\JSE\JDBC\Programs>set classpath=C:\Program Files (
x86)\Java\jdk1.7.0\bin;.;%classpath%

D:\Technology\Java Technology\JSE\JDBC\Programs>java JDBCUsingType1
java.lang.ClassNotFoundException: sun.jdbc.odbc.JdbcOdbcDriver

D:\Technology\Java Technology\JSE\JDBC\Programs>set path=C:\Program Files (x86)\
Java\jdk1.7.0\bin;.;%path%

D:\Technology\Java Technology\JSE\JDBC\Programs>java JDBCUsingType1
Driver loaded...

D:\Technology\Java Technology\JSE\JDBC\Programs>
```

So after loaded the type 1 driver now

We have to establish the connection.

DriverManager

To establish the connection we have to take the help of DriverManager class which contain the getConnection() static method which takes totally three parameter
first one is url, second one is username and third one is password.

The DriverManager class getConnection method return type is Connection interface reference.
The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Connection
Where Connection interface reference represents the session between your java application and the database.
A connection is the session between java application and database. The connection interface is a factory of Statement, PreparedStatement , CallableStatement and DatabaseMetadata. Connection interface provide many methods for transaction management like commit() or rollback() etc.

Statement

Then using the Connection interface reference we have to create the Statement interface reference which is used to execute the static SQL statement and return the result set. There are two other sub interfaces PreparedStatement, which extends Statement and CallableStatement, which extends PreparedStatement.
Statement interface provides method to execute queries with the database. The statement interface is a factory of ResultSet

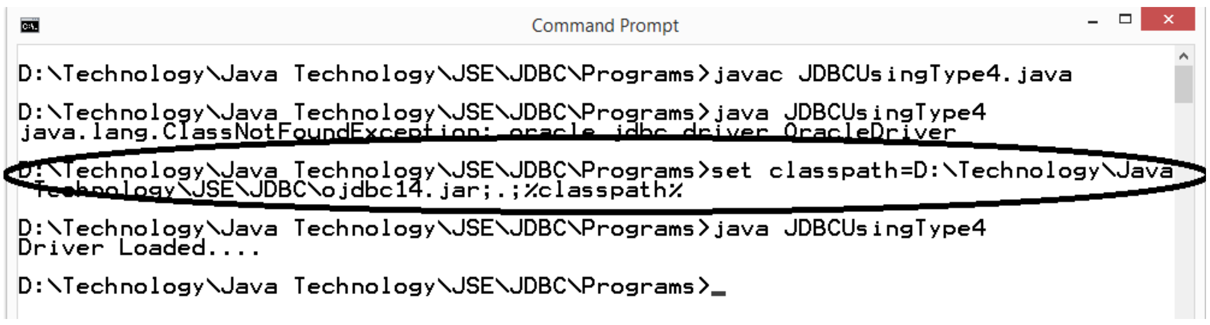
ResultSet is an interface that manages the resulting data returned from the Statement. which provide the set of method which help to move the cursor forward direction and get the specific cell value.

Connecting database using the Type 1 Driver for the Oracle database.

Error generating in the window 8 for the DSN.

Connecting database using the Type 4 Driver for the Oracle database.

When you are using type 4 driver for Oracle or MySQL we have to set the classpath mandatory temporary or permanent.



```
D:\Technology\Java Technology\JSE\JDBC\Programs>javac JDBCUsingType4.java

D:\Technology\Java Technology\JSE\JDBC\Programs>java JDBCUsingType4
java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver

D:\Technology\Java Technology\JSE\JDBC\Programs>set classpath=D:\Technology\Java
 Technology\JSE\JDBC\ojdbc14.jar;.;%classpath%

D:\Technology\Java Technology\JSE\JDBC\Programs>java JDBCUsingType4
Driver Loaded....

D:\Technology\Java Technology\JSE\JDBC\Programs>_
```

JDBCUsingType4.java

```
import java.sql.*;
class JDBCUsingType4 {
	public static void main(String args[]) {
	try {
	Class.forName("oracle.jdbc.driver.OracleDriver");
	System.out.println("Driver Loaded....");
	Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521/XE","hr","tiger");
	System.out.println("Establish the connection...");
	Statement stmt = con.createStatement();
	System.out.println("Statement created...");
	ResultSet rs = stmt.executeQuery("select employee_id,first_name,salary from
employees where employee_id between 100 and 110");
	while(rs.next()) {
	System.out.print("Id
"+rs.getInt(1)+"\tName="+rs.getString(2)+"\tSalary"+rs.getFloat(3));
	System.out.println();
	}
	rs.close();
	stmt.close();
	con.close();
	}catch(Exception e) {
		System.out.println(e);
	}
	}
}
```

PreparedStatement

Statement interface is use to execute only simple SQL statement like CRUD (create, retrieve, update and delete). These query are static query must be known as compile time only. Where PreparedStatement used to executing dynamic SQL statements. PreparedStatement accepts runtime values and these values can be passed as parameters (?) as the runtime using the methods setXXX().

PreparedStatement query (but for parameters) is compiled and placed in the database cache. For this reason, PreparedStatement is known as "partially compiled statement". At runtime, only the parameter values are compiled and inserted into the partially compiled query and executes and thereby PreparedStatement gives higher performance than Statement interface.

ResultSetMetaData interface

The metadata means data about data ie. we can get further information from the data.

If you have to get the metadata of a table like total number of column, column name, column type etc. ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

How to get the object of the ResultSetMetaData

```
ResultSetMetaData rsmd = rs.getMetaData();
```

DatabaseMetaData interface

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, number of tables etc.

CallableStatement

The CallableStatement is used to execute stored procedure as well as functions.