Supplementary material for Chapter 2 of the McGraw Hill book: “Object Oriented Software Engineering: Practical Software Development Using UML and Java”

The Basics of Java

In the first part of this chapter we have reviewed some of the main principles of object orientation. In this section we review the Java programming language and show how its object-oriented features are implemented. This section is not intended to give comprehensive coverage of Java; it is instead intended to serve as a review for those who have taken a first course in Java, and to assist those who learned an OO language other than Java to understand the examples later in the book.

Java, like C++, shares much of its syntax with the C programming language. If you know how to write statements, including variable declarations and loops, in one of these three languages, then you have a head start in learning the others. However, many other details differ among the languages. In particular, the data types and libraries available are considerably different.

Java was developed at Sun Microsystems. It has a useful combination of features that, combined with the fact that it is a member of the C/C++ family, have made it very popular in recent years:

- **Platform independence**: Java is designed to be run using a *virtual machine*, or *VM*. Java compilers compile source code (typically found in files ending with the `.java` suffix) into files containing *bytecode* (typically in files ending with the `.class` suffix, or in libraries ending with `.jar`). Bytecode is like a universal machine language – it is very low-level in the sense that it is not designed to be read by human beings. At the same time, programs in bytecode can be run on any computer that has a VM. The VM acts as an *interpreter* for the bytecode; this makes Java programs portable.
- **Use over the Internet**: Java is designed with very easy-to-program networking capabilities. In addition, small programs called applets can be loaded directly into a web browser.
- **Security checks**: A properly configured Java VM will not allow violations of certain security constraints when programs are downloaded over the Internet.
- **Removal of troublesome C++ features**: C++ has many powerful capabilities, such as pointer arithmetic, multiple inheritance, macros and operator overloading. Although powerful, these can make programs difficult to understand, so they have been deliberately left out of Java. Also, Java will not let programmers refer to information beyond the end of an array, or follow an invalid pointer. In C and C++, these situations frequently cause programs to crash.
- **Garbage collection**: You do not need to free objects from memory when you no longer need them. The VM will reclaim objects that is no longer being used.

Java tends to be less efficient than programming languages like C or C++ for two reasons: Firstly, Java’s safety checks and garbage collection can slow down execution. Secondly, the interpretation of bytecode is not as fast as direct execution of machine code. Most virtual machines mitigate this latter problem by using what is called *just-in-time compilation* (JITC); this means that the first time a method is executed, the VM converts it into machine code and stores the machine code to save work on subsequent calls. However, even with JITC, Java is not as fast as a fully compiled language.

However, the reduced efficiency of Java is often not a problem. In today's world, hardware is so fast that efficiency does not always need to be a high priority. Also, the cost of programmers' time tends to be far more expensive than hardware, so development teams can save money by using Java, which is easier to program than C++. The savings in programmer time can easily pay for faster CPUs.

Nevertheless, Java's lower efficiency means that it is not suited to every application: A program that primarily performs CPU-intensive calculations should probably be written in a more efficient language.

Incidentally, the fact that Java uses a virtual machine or has garbage collection is not unique. Smalltalk is another OO language that also has these features; in fact many of the ideas in Java were adopted from Smalltalk.

One more point about Java before we start discussing its details: There is a related language called JavaScript which is used to add functionality to web pages. Despite sharing much of the same syntax and many of the same keywords, Java and JavaScript should be seen as clearly separate languages.

The simplest statements

An assignment statement in Java uses the '=' symbol; a semicolon terminates statements:

```
aVariable = 5;
```

A call to a procedure (a method) in the *current* class looks like the following:

```
resultVariable = methodName(argument1, argument2);
```

In this case, the result is assigned to the variable **`aVariable`**.

If there are no arguments to a method, use open and close parentheses with nothing in between, thus:

```
resultVariable = noArgumentMethod();
```

A call to an instance method of the object in variable **`b`** looks like the following:

```
resultVariable = b.methodName(argument);
```

The dot (.) symbol is also used to access an instance variable of an object stored in a variable, thus:

```
aVariable = b.variableName;
```

It is considered better design practice, however, to avoid directly accessing instance variables in this manner. Instead, try to obtain the same information using an instance method of the object in variable **b**, as in the previous example.

Several statements can be placed together in a block, surrounded by braces ('**{**' and '**}**', also colloquially known as curly brackets). Blocks are used primarily as the bodies of classes, loops, conditional statements and for exception handling; all of these uses are discussed later. Here is an example of a block:

```
{
   a = 5;
   b = computeSomething(c);
}
```

A simple instance method

A method in Java looks like the following example. The heading of the method has these components:

- It can be declared **`public`**, **`protected`** or **`private`**; we will discuss these keywords later.
- It has a return type, which may be **`void`**, indicating that the method does not return anything.
- It has a pair of parentheses which can enclose a list of formal argument declarations, separated by commas.

```
public double credit(double amountToCredit)
{
  balance = balance + amountToCredit;
  return balance;
}
```

Comments

Comments in Java take two forms: Two slashes '**`//`**' indicate that the rest of the line is to be considered a comment and hence ignored by the compiler. Alternatively you can start a comment with '**`/*`**' and end it with '**`*/`**'; anything between this pair of markers is ignored.

Variable declarations and basic data types

You declare variables in Java by giving the data type followed by the name of the variable. A variable declaration can be placed practically anywhere in Java code, although it is good practice to only declare variables at the beginning of blocks. The following are some examples of variable declarations, illustrating the *primitive* Java data types.

```
byte aByte;             // an 8-bit value
short aShort;           // a 16-bit integer
int anInteger;          // a 32-bit integer
long aLong;             // a 64-bit integer
float aFloat;           // a floating point number
double aDouble;         // a double-precision floating-point number
char aChar;             // a character of Unicode, discussed below
boolean aBoolean;       // must be one of true or false
```

In the above declarations, the types all start with a lower-case letter. This distinguishes them as primitive data types. You can use primitive data types for many purposes, but variables declared using these types do *not* contain objects – their contents are not instances of any class.

The set of operations available to work with primitive data types is rather limited. The basic arithmetic operators **`+`**, **`-`**, **`*`**, **`/`** and **`%`** (for modulus) can be used with values that have type **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, or **`double`**.

The logical operators **`&&`** (and), **`||`** (or) and **`!`** (not) can be used to operate on **`boolean`** values.

Sidebar: Short circuit operators

The **`&&`** operator has an interesting property. In the expression **`a() && b()`**, Java would not even bother to compute **`b()`** if **`a()`** turned out to be **`false`**. The entire expression would immediately return **`false`** as soon as the falseness of **`a()`** is determined. The value of **`b()`** would only need to be calculated if **`a()`** is, in fact, **`true`**. The **`&&`** operator is therefore called a *short circuit* operator. It is important that the programmer be aware that the right-hand side of the expression might never be computed.

Similarly, **`||`** is also a short circuit operator. Its right-hand side is not computed if the left-hand side returns **`true`**.

In addition to manipulating primitive values, variables are also used to manipulate *objects* in a Java program. To declare such variables, you use the name of a *class* as their type, indicating that an instance of that class is to be put into the variable. Whenever the type in a variable declaration starts with a capital letter, you know that it is a class, not a primitive type.

Java defines many standard classes that can be used in any program. Important examples are **`String`** and **`ArrayList`**, shown below. You can also define your own classes such as **`PostalCode`**.

```
String aString;            // an object that contains a string of
                           // characters (discussed later)
ArrayList anArrayList;     // an object that contains a list of
                           // objects (discussed later)
PostalCode aPostalCode;    // an object of class PostalCode
```

To work with variables such as these, you have to call methods or access instance variables that are found in the declared class or its superclasses. You cannot use the arithmetic operators.

The **`==`** operator, which returns a **`boolean`**, is used to compare any two items which could be expression results, simple values or the content of variables. The operator tests if the items are *identical*, which means they either refer to the same objects or have the same primitive values. A common bug is to use the identity operator **`==`** when the assignment **`=`** was meant, or vice versa. This bug will normally be caught by the compiler, but not if you are working with **`boolean`** items.

To test whether two variables contain objects that are *equal* (i.e. contain the same data, or *state*, but are not necessarily the same object), you call the **`equals`** method using an expression such as the following:

```
boolean b = aPostalCode.equals(anotherPostalCode);
```

In some situations in Java, you want to do something with primitive values beyond what the basic operators described above can accomplish. Java therefore provides a set of classes, called *wrapper* classes, corresponding to each of the primitive types. Each instance of these classes simply contains an instance variable of the corresponding primitive type. The following are sample declarations:

```
Integer anIntegerObject;
Float aFloatObject;
Double aDoubleObject;
Byte aByteObject;
Character aCharacterObject;
Boolean aBooleanObject;
```

It is very easy to confuse instances of these classes with their primitive equivalents, so it is important to pay careful attention to where a primitive is required and where an object is required. When you work with instances of these classes, you cannot use the ordinary arithmetic or logical operators, instead you have to use methods.

For example imagine you had two instances of class **`Integer`**, called **`integer1`** and **`integer2`**. If you wanted to add them and store the result back into **`integer1`**, you would have to write the following rather inconvenient statement:

```
integer1 = new Integer(integer1.intValue() + integer2.intValue());
```

Due to this complexity, and the inefficiency of the method calls, arithmetic in Java is normally done using primitive values instead of instances of the wrapper classes.

Class variables and class methods

A *class variable* in Java is a variable that is marked **`static`**, declared in the body of the class (not inside a method).

You access a class variable by specifying the name of the class, followed by a dot, followed by the name of the variable. An example is:

```
Color.blue
```

Similarly, a *class method* is a method marked **`static`**. When a class method executes it is not working on a particular instance of the class. You can therefore only manipulate class variables or call class methods of the class – i.e. you cannot manipulate a class's instance variables or call its instance methods.

You call a class method by using the name of the class, followed by a dot, followed by the name of the method (the name of the class can be omitted when calling a class method in the current class).

The wrapper classes have many useful class methods such as the following:

```
int i = Integer.parseInt(aString); // converts aString to an int
String s = Integer.toHexString(anInt); // converts to hexadecimal
boolean b = Character.isDigit(aChar); // tests if the character is a digit
```

Java documentation

One of the most important skills for anyone designing or programming software is to be able to navigate the documentation and look up the methods available to achieve some objective. Java comes with extensive on-line documentation about each class and method; you should become familiar with how to look up information in this documentation.

The documentation is available on Sun's web site – see the 'For More Information' at the end of the chapter for details. You may also find a copy of the documentation that was installed with the Java compiler and VM on your local disk or network.

♦ **Exercise**

E 1 Find the methods in the Java documentation that do the following:

a) Convert a **`boolean`** to a **`Boolean`** and vice-versa.
b) Convert an **`int`** to an **`Integer`** and vice-versa.
c) Convert a **`String`** to a **`double`** (i.e. parse the **`double`** value contained in a **`String`**) and vice-versa.
d) Convert an **`int`** to a **`double`**.
e) Find out what a hash code is and how to compute the hash code of a **`String`**.
f) Find out what a **`Vector`** is and how to compute the number of elements it contains.

Special operators and operator precedence

Java, like C and C++, has some special operators that are widely used to shorten certain expressions. The most important of these are shown in Table 1.

There is a strict order of precedence that Java uses for evaluation of expressions. The precedence rules for the operators used in this book are described in Table 2. Operators at the top take precedence over operators lower in the table. At the same level of the table, operators are evaluated from left to right.

Conditional statements and choice among alternatives

A *condition* in Java is a statement that evaluates to a **`boolean`** value (**`true`** or **`false`**). The following are examples of conditions:

```
aNumber > 5
aNumber < 5
aNumber > 5 && anotherNumber < 7
aNumber == anotherNumber
```

Table 1: Special operators in Java

Operator	**Example expression**	**Equivalent longer expression**
`++` postfix form	`a++;`	`a=a+1;`
	`b=a++;`	`b=a; a=a+1;`
	`++a;`	`a=a+1;`
`++` prefix form	`b=++a;`	`a=a+1; b=a;`
`--` postfix form	`a--;`	`a=a-1;`
	`b=a--;`	`b=a; a=a-1;`
	`--a;`	`a=a-1;`
`--` prefix form	`b=--a;`	`a=a-1; b=a;`
`+=`	`a+=b;`	`a=a+b;`
`*=`	`a*=b;`	`a=a*b;`
`-=`	`a-=b;`	`a=a-b;`
`/=`	`a/=b;`	`a=a/b;`

Table 2: Precedence of the most important operators in Java.

Operators	Comments
`()` `[]`	Anything in parentheses, including array indexing, is evaluated first
`++` `--` `!`	Unary operators
`*` `/` `%`	Multiplicative operators
`+` `-`	Additive binary operators
`>` `>=` `<` `<=`	Relational comparison operators
`==` `!=`	Identity comparison operators
`&&`	Logical AND
`||`	Logical OR
`?:`	Ternary if-then-else, discussed below
`=` `+=` `*=` `-=` `/=`	Assignment

There are three ways in Java to use conditions in order to make choices among alternative code to execute: **`if`**, **`?:`**, and **`switch`** statements. The **`if`** statement has the following form:

```
if(condition)
{
   // statements to execute if condition is true
}
else
{
   // statements to execute if condition is false
}
```

The **`else`** block can be omitted if there is nothing to do in the false case.

If there is only a single statement to be executed in the true or false case, then the curly brackets can be omitted. However leaving the curly brackets can often make the code clearer.

The **`?:`** operator can also be used to execute one of two alternative expressions, depending on the value of a condition:

```
result = (condition) ? doSomething() : doSomethingElse();
```

If **`condition`** is **`true`**, then **`result`** is set to the expression following the question mark, otherwise **`result`** is set to the expression following the colon. The **`?:`** operator can shorten some code, but make other code harder to understand. As a rule of thumb, always choose the form which results in the most readable code.

A **`switch`** statement has the following form:

```
switch(primitiveVariable)
{
   case value1:
      // statements to execute if primitiveVariable equals value1
      break;
   case value2:
      // statements to execute if primitiveVariable equals value2
      break;
   ...
   default:
      // statements to execute if none of the above is true
      break;
}
```

A few general comments about the **`switch`** statement:

- The **`break`** labels are important: If the **`break`** in the **`value1`** case was omitted above, then whenever **`value1`** occurred, the statements for *both* **`value1`** and **`value2`** would be executed.
- The **`default`** case is executed when the value in **`primitiveVariable`** is something other than one of the explicit cases.

You should use polymorphism to reduce the need for **`switch`** statements.

Loops

There are two main types of loops in Java, **`for`** and **`while`**; their syntax is identical to loops in C and C++. A **`while`** loop has the following structure:

```
while(condition)
{
   // statements to keep executing while condition is true
}
```

A **`for`** loop has the following structure:

```
for(initializer; condition; incrementer)
{
   // statements to keep executing while condition is true
}
```

The *initializer* is a simple statement that sets up some kind of initial condition for the loop – often initializing a variable. The *condition* is a statement that returns a **`boolean`** value, normally testing the variable initialized in the initializer; the condition is evaluated before every iteration through the loop. The incrementer is a statement executed after every iteration through the loop, typically updating the variable set in the initializer.

You can, in general, interchange a **`while`** loop and a **`for`** loop. To turn a **`while`** loop into a **`for`** loop, move the initializer before the **`while`** statement, and ensure that the incrementer is the last statement executed in every iteration. The advantage of using a **`for`** loop is that all the information about controlling the loop is kept in one place; the disadvantage is that it can be slightly harder to read the code of a **`for`** loop.

Examples of loops can be found in the example code later in this chapter.

Overall structure of a class

All the code of a Java program must be placed inside classes. This is an important difference from C++, which allows some code to exist outside classes. You put each Java class in a file of the same name. An exception to this is in environments like IBM VisualAge for Java which do not organize code into files, but instead keep the code in a *repository*.

The overall structure of a class should look something like the following:

```
class classname
{
   // declarations of variables

   // declarations of constructors (discussed below)

   // declarations of other methods with public ones first
}
```

The exact order of these elements is a matter of style.

Constructors and the creation of objects

Constructors are procedures that are called whenever a new object is created. Each constructor has the same name as the class, but can have different sets of arguments. The purpose of a constructor is to initialize the instance variables of a newly created object and perform any other needed initialization.

The following are two constructors that might be used in a class **`Account`**. The first sets the balance to a specific initial value, whereas the second, lacking a second argument, sets the balance to zero.

```
public Account(String accountHolder, float initialBalance)
{
  this.accountHolder = accountHolder;
  balance = initialBalance;
  opened = Calendar.getInstance();
}

public Account(String accountHolder)
{
  this.accountHolder = accountHolder;
  balance = 0.0;
  opened = Calendar.getInstance();
}
```

Both of the above constructors initialize three instance variables. The value **`this`** represents the *current object*. It is being used here to distinguish between the instance variable **`accountHolder`** and each constructor's argument of the same name.

You use the **`new`** operator to create a new object. This operator sets aside memory for the object and calls a constructor. The following are two illustrations of the use of new:

```
String accountHolder = “Tim”;
float initialDeposit = 100.0;
acct1 = new Account(accountHolder, initialDeposit);
acct2 = new Account(accountHolder);
```

The constructor chosen is the one that has the same argument types as the arguments that follow the **`new`** operator. Therefore, when **`acct1`** is created, the first constructor (the one with a string and a float argument) would be called, whereas **`acct2`** would be constructed using the second constructor. A constructor may have no arguments at all.

A new object can be created by using the **`new`** operator in a variable declaration. An object is then created whenever execution enters the particular block that contains that declaration, or, in the case of an instance variable, when an instance is created. The following gives an example:

```
Account anAccount = new Account(holderName);
```

It is important to remember that a constructor of class **`Account`** is called in the above statement. The particular constructor chosen will depend on the class of **`holderName`**.

Alternatively, the declaration of the variable can leave it un-initialized – i.e. it is left to be initialized on a later line. A Java compiler should, however, warn you if your program can execute code that accesses un-initialized variables.

```
Account anAccount;
…
anAccount = new Account(holderName, initialBalance);
```

An instance variable that is declared with an object type, but that is not yet initialized, has the primitive value **`null`**.

Arrays

An array variable in Java is declared using square brackets following the type. The following are some examples:

```
int[] anIntArray = new int[25];
byte[] aByteArray;  // not initialized
Account[] anAccountArray = new Account[numAccounts];
```

As these examples show, arrays can be composed both of primitive types like **`int`** and **`byte`**, and also of instances of classes, such as **`Account`**. The number of elements in an array can be a constant or a variable.

Arrays have a special status in Java; they are objects, but they are not instances of classes which you can subclass or for which you can write your own code.

In order to access an element of an array, you use square brackets and specify an index. You can also request the length of an array. For example, the following sums all the elements of an **`int`** array:

```
int sum=0;
for(int i=0; i<anIntArray.length; i++)
{
   sum += anIntArray[i];
}
```

You should generally minimize your use of arrays, and you should always avoid them if you do not *a-priori* know the number of items it will contain. Unfortunately, programmers often hard-code a maximum size, which makes programs inflexible and bug-prone.

The alternatives to arrays are the classes which we will discuss in the next two sections: **`String`**s, for collections of characters, and collection classes such as **`Vector`** and **`ArrayList`**, for collections of arbitrary objects.

Arrays have the advantage of being more efficient than these specialized classes; however, the specialized classes have wide variety of useful operations, and some of them have the ability to grow as new objects are inserted. In addition, programs written using the specialized classes are often easier to read than programs which uses arrays.

An important thing to remember about arrays (and the specialized collection classes) is that they are *zero-based*. This means the first element is element 0. It also means that the highest numbered element is one less than the length of the array.

Characters and Strings

In many programming languages, a character is an 8-bit byte encoded using ASCII. Unfortunately, although ASCII was an excellent invention for the English-language applications of the 1950's and 1960's, it is not capable of representing the wide variety of printed symbols used in other languages.

To make Java extendible to most of the written languages of the world, it uses a coding scheme called *Unicode* instead of ASCII. Characters in Unicode are not restricted to one byte; however the exact details of the representation of each character is normally not important to programmers. All a programmer needs to know is that when he or she is working with characters, they could be from an arbitrary character set. The basic ASCII characters remain a part of Unicode, and programmers can still use the **`byte`** datatype to work with real ASCII characters if they truly need to do so. It is bad programming practice, however, to use bytes for textual data which is to be exposed to the end user.

Strings in Java are collections of characters; the **`String`** class provides a rich set of facilities for manipulating such objects. Some facilities for dealing with strings are also built into Java at a primitive level. In particular, you can define a string constant by placing it in double quotes, and you can concatenate two strings by using the **`+`** symbol. The following are some simple examples of string manipulations:

```
“Insert a variable (” + aVariable + “) between two constant strings”
```

Note that **`aVariable`** above could contain *anything*. Any object can be converted into a string using a **`toString`** method. In the above example the **`toString`** method would be invoked on **`aVariable`** to generate an instance of **`String`**. It is good practice to write your own **`toString`** method in every class; it should return a string that will help identify each instance. The default implementation of **`toString`** outputs the name of the class followed by a hexadecimal code that distinguishes one instance from another.

The following two statements illustrate one of the many operations available to work with strings in Java. Note that as with arrays, the first character in a string is at index 0. For the **`substring`** operation, the first argument is the starting position, and the second argument is the ending position +1; the result is a new **`String`** with a sequence of characters from the original string.

```
String sub = “submariner”.substring(0,3);  // = “sub”
String marine = “submariner”.substring(3,9); // = “marine”
```

♦ **Exercise**

E 2 Use the Java documentation to search for the following information regarding the class and **`String`**.

a) What happens if you call the **`substring`** operation with only one argument?

b) How do you remove trailing white space from a **`String`**?

Collection classes and their iterators

We have seen how you can use arrays to create fixed-size collection of objects or primitive data items. We have also seen how **`String`**s provide collections of **`Character`**s. Java also provides a variety of other classes for working with collections of objects. Only the most important ones are shown below; you are invited to study the documentation to learn more about them.

- **`ArrayList`**: This allows you to build collections of objects that grow as more objects are added. Important methods include **`set`**, **`add`** and **`remove`**.
- **`Vector`**: This class is like **`ArrayList`**, with some subtle differences that we will not discuss here. **`Vector`** has been around since Java was first released, whereas **`ArrayList`** is newer.
- **`LinkedList`**: Another class that has many of the same functions as **`ArrayList`** and **`Vector`**, except that it is more efficient for certain operations, e.g. inserting an element in the middle, and less efficient for other operations, e.g. extracting an arbitrary element.

A common operation with collection classes is to do something with every member of the collection. Java provides a class called **`Iterator`** to do this. To create an iterator, you simply use the methods called **`iterator`** or **`listIterator`** found in any collection class. Then you can repeatedly call the method **`next`** to obtain successive elements. The method **`hasNext`** enables you to find out if there are any more elements. The following example counts the number of empty strings in a collection of strings. We will discuss the notation '**`(String)`**' in the next section.

```
emptyCount = 0;
Iterator iter = aCollection.iterator();
while(iter.hasNext())
{
  if(((String)iter.next()).length()==0)
    emptyCount++;
}
```

Iterators also have a **`remove`** method, that allows you to selectively delete elements of the underlying collection.

In older code (prior to Java version 1.2), you may see the use of **`Enumeration`**. This should be avoided now since it is more cumbersome to use, yet provides similar functionality to **`Iterator`**.

Casting

The block of code in the last subsection illustrated an important issue. The **`next`** operation of an **`Iterator`** declares that it returns an **`Object`**, the class that is considered to be the ultimate superclass of all other classes. What this means is that when **`next`** is executed, the object returned can be of *any* Java class – it all depends on what was originally put into the underlying collection.

However, if you put the result of **`next`** into a variable of type **`Object`**, you could only invoke those few operations defined in class **`Object`**. So you have to use a mechanism called *casting*. Casting works when you, as a programmer, know that the object in a variable (or the return type of an expression) is actually a *subclass* of the declared type of that variable or expression.

To cast a variable or expression you precede it with the name of the resulting class, surrounded by parentheses, like this: **`(String)i.next()`**. This statement is a bit like making a contract of the following form: "I, the programmer, know that the **`next`** method, in this *particular* case, is really going to return a **`String`**, even though **`next`** is declared to return type **`Object`**. So, trust me, compiler, and let me use the result as if it were a **`String`**. I agree to pay the consequences if I am wrong: an error will occur at run time." The type of error that occurs is the raising of a **`ClassCastException`**. We discuss exceptions below.

♦ **Exercise**

E 3 Write the necessary expressions to do the following: Create an **`ArrayList`** of arbitrary **`String`**s (which you can hard-code, even though this would be bad practice in a real system). Then use an **`Iterator`** to find the sum of the lengths of the **`String`**s, as well as the longest **`String`** and the shortest **`String`**.

Exceptions

When something goes wrong in the execution of a program, such as an attempt to divide by zero, Java *throws an exception*. Throwing an exception means that instead of executing the next line of code, Java looks for some code to *handle* the exception and executes that instead. Java programmers are responsible for anticipating things that can go wrong and writing exception handling code in preparation. The **`try-catch`** construct provides the basic capability for this:

```
try
{
   result = numerator / denominator;
   validResult = true;
}
catch (ArithmeticException e)
{
   validResult = false;
}
```

Any division by zero that occurs when executing the **`try`** block will result in execution of the **`catch`** block. Once either block completes, execution continues at the statement after the catch block.

What happens if an exception is thrown in a statement that is not in a **`try`** block with an appropriate **`catch`** statement? The answer is that Java will look to see if the *caller* of the current method is within a **`try`** block that has an appropriate **`catch`** block. It will continue to look at callers right up to the main program, and will finally report an error if it has not been able to find any suitable **`catch`** block.

If you are writing code in a class that could throw an exception, and you do not want to write a **`try`**-**`catch`** block, but want to rely on the caller of the method to catch the exception, then you have to do something special: At the start of the method definition you have to declare that you are *not* handling certain exceptions by listing them in the following manner:

```
int methodThatMayDivideByZero()
   throws ArithmeticException
{
   // code for the method that could throw the exception
}
```

Java provides many types of built-in exceptions. Each is, in fact, a class. When an exception is raised, an instance of that class is created that contains information about the problem causing the exception.

You can also create your own exception classes representing things that can go wrong in the computations your code performs. For example, in a banking application you might decide to define an exception called **`OverdraftLimitException`**. You could then explicitly throw an exception in code that might result in exceeding an overdraft limit. The following illustrates how this might be done. As with any Java classes, each of the following classes should be in a separate file.

First, here is the new exception class. (This is also your first exposure to creating a *subclass*, using the **`extends`** keyword; we will discuss this in more detail shortly).

```
class OverdraftLimitException extends Exception
{
}
```

Now, here is the class that uses the exception:

```
class Account
{
  MoneyAmount overdraftLimit;
  MoneyAmount balance;
  …
  MoneyAmount debit(MoneyAmount debitAmount) throws OverdraftLimitException
  {
    if(debitAmount > balance + overdraftLimit)
      throw new OverdraftLimitException();
    balance -= debitAmount;
    // perhaps do other stuff here
  }
}
```

♦ **Exercise**

E 4 Write try-catch blocks to handle the following situations. You will need to refer to the Java documentation to discover the appropriate exception names to use.

a) You attempt to transform the **`String`** “1A” into an **`int`**.
b) You attempt to create an array of size –3.
c) You attempt to access an array at index 7 where its length is 7.

Simple terminal I/O

Most serious programs interact with the user using graphical user interfaces. We will give some tips for how to do this in Chapter 7.

It is still useful, however, to know how to read and write information from the console (e.g. the DOS console or a Unix terminal). The following are examples of the basic statements you need to use.

Java’s basic terminal output statement is rather simple and takes the following form,. Any **`String`** can be an argument to **`System.out.println`**, including strings constructed using the concatenation operator **`+`** described earlier.

```
System.out.println(“This line will be printed”);
```

Java’s mechanism for inputting what the user types at the console is less elegant. As the following code shows, you have to first create a **`byte`** array of sufficient size. You then call **`System.in.read`**, which waits for the user to hit a carriage return and then places what was typed into the byte array. Finally you have to convert this into a **`String`**, trimming off any whitespace that may have been unexpectedly added.

```
byte[] buffer = new byte[1024];
System.in.read(buffer);
String theInput = new String(buffer).trim();
```

If you wanted to interpret the input as something other than a string, you could write statements like this:

```
float aFloat = Float.valueOf(theInput).floatValue();
```

We will leave it up to you to look up the necessary methods to convert to other data types.

Generalization and inheritance

To create a subclass in Java, you use the **`extends`** keyword, as in the following example:

```
public class MortgageAccount extends Account
{
   // body of the class
}
```

According to the above, class **`MortgageAccount`** is a subclass of **`Account`**. Any instance variables or methods defined in **`Account`** (or its superclasses) are now also implicitly present in the new subclass – in other words they are *inherited*.

In Java, a class can have only one superclass – this is called *single inheritance*. Other languages, such as C++ allow more than one parent. Multiple inheritance can result in more complex systems, hence the designers of Java decided it was better to allow only single inheritance.

Java does, however, provide a mechanism, *interfaces*, which provide the important benefits of multiple inheritance without the drawbacks. We will discuss these shortly.

Abstract methods and abstract classes

To create an abstract method in Java, you simply mark it **`abstract`**. You must not write any executable statements in the body of the method; the body is simply omitted. The method serves as a placeholder, indicating that subclasses must have concrete implementations.

Similarly, you declare a *class* to be abstract by specifying the **`abstract`** keyword on the first line when you declare the class.

Interfaces

An interface in Java is like a class except that it does not have any executable statements – it only contains abstract methods and class variables.

An interface differs from an ordinary abstract class in an important way: It *cannot* have any concrete methods or instance variables, whereas an abstract class can.

The value of an interface is that it *specifies* a set of methods that a variety of different classes are to implement polymorphically. The classes that implement the interface do not have to be related to each other in any other way.

A class uses an **`implements`** clause, as in the example below, to declare that it contains methods for each of the operations specified by the interface. In Java, a class can implement more than one interface, whereas it can only extend one superclass. As mentioned above, this is quite different from languages like C++.

You can declare a variable to have an interface as its type. This means that, using the variable, you can invoke any operation supported by the interface. Dynamic binding will occur so that the correct method is run.

For example, the following code specifies that any class can implement the **`Ownable`** interface. Furthermore a class that implements **`Ownable`** must provide concrete implementations for both of the operations. The code for the **`Ownable`** interface would be put in a separate file by that name, just like a class.

```
public interface Ownable
{
  public abstract String getOwnerName();
  public abstract void setOwnerName(String name);
}
```

Here are examples of how classes might specify that they implement this interface.

```
public class BankAccount implements Ownable
{
  …
  public String getOwner()
  {
    return accountHolder;
  }
  public void setOwner(String name)
  {
    accountHolder = name;
  }
…
}

public class Pet extends Animal implements Ownable
{
  …
  public String getOwner()
  {
    return owner;
  }
  public void setOwner(String name)
  {
    owner = name;
```

```
   }
   …
}
```

The following shows how you could now declare a variable that can contain either a **`Shape2D`** or a **`Person`** You can then ask for the image of whatever is in the object.

```
Drawable aDrawableObject;
…
aDrawableObject.drawImage()
```

♦ **Exercise**

E 5 Look in the Java documentation to discover the methods available in classes that implement the following interfaces:

a) **`Comparable`**
b) **`Collection`**
c) **`Shape`**
d) **`Runnable`**
e) **`Cloneable`**
f) **`Iterator`**

E 6 Java has an interface called **`Shape`**. This is implemented by a variety of classes in different hierarchies. Study **`Shape`** and its implementing classes to determine how it differs from the **`Shape2D`** hierarchy presented in Figure **Error! Reference source not found.**. What are the advantages and disadvantages of the two approaches?

Packages and importing

A *package* in Java is used to group together related classes into a subsystem. Each package is given a name composed of a series of words separated by dots. For example **`java.lang`** is one of the important packages of classes that is part of standard Java.

All the classes which belong in one package have to be put into a directory with the same name as the package. The components of a package name that come first correspond to higher-level directories. For example if you created a package called **`finance.banking.accounts`**, you would put that in a directory called **`accounts`**, which would be in a directory called **`banking`**, which would be in a directory called **`finance`**. A further convention, not always adhered to, is to prepend to the package name the domain name of the organization, with the components inverted. So, for example if **`mcgrawhill.com`** owned the package **`finance.banking.accounts`**, then the full package name might be **`com.mcgrawhill.finance.banking.accounts`**. This assures that each package name is unique in the world.

A file containing a class should always declare the package to which it belongs using the **`package`** keyword, thus:

```
package finance.banking.accounts;
```

If a class wants to use the facilities of another package, its file should contain an **`import`** statement, such as the following.

```
import finance.banking.accounts.*;
```

By importing a package, you are saying that all the code in that class file knows about the classes in the imported package – in other words you can refer to the classes in the imported package by name.

A package therefore defines what is often called a *name space*; the total name space of any class file includes the names in the file's own package plus the names in all imported packages.

It is possible for two classes to have the same name as long as the names do not clash – i.e. the identically named classes are not in the same package – and their packages are never both imported into the same file. Despite this rule, however, it is a good idea to try to avoid giving two classes the same name, since somebody in the future might want to import both packages and hence create a name clash. If you ever do encounter a name clash, you can resolve it by qualifying the name of a class with the name of its package, for example, if there were two **`Account`** classes in the packages you were importing, you could write an expression like this:

```
mybank.Account newAccount = new myBank.Account(accountHolder);
```

♦ **Exercise**

E 7 Java has many important packages that programmers use heavily. Study the contents of the following packages, using the Java documentation.

a) **`java.lang`**
b) **`java.util`**
c) **`java.math`**
d) **`java.io`**

Access control and scope: public, protected and private

By default, the methods and variables in a class can be accessed by methods in any class in the same package. This default rule is insufficient in two cases, however:

- Sometimes you want to restrict access to variables or methods. If you know that many methods access a given variable, then changing the variable's definition becomes difficult because you may have to change all places where it is accessed. If you had restricted access to the variable when you first developed the system, then it would later be easier to change the variable. This is because it would be accessed from fewer places so it will be easier to find out where those places are.
- Sometimes you want just the opposite: You want to create a list of methods that are widely available for use by methods outside the current package.

When you define any method, instance variable or class variable, you can precede the definition with the keywords **`public`**, **`protected`** or **`private`** to control exactly what code can have access to the method or variable. Table 3 shows the effect of each of the keywords, starting with the least restrictive access and moving to the most restrictive.

It is good practice to restrict access to methods, instance variables and class variables as much as possible. This is in line with the concept of information hiding presented earlier: You want the details of the implementation of a class, class hierarchy or package to be hidden as much as possible from outsiders. This makes code easier to understand and change, and also tends to make designs more flexible. When we discuss higher level design in Chapter 9, we will see that restricting access reduces what we will call *coupling*, which is the interconnection among various components of a system.

Some simple rules for access control are as follows: Make all instance variables as private as reasonably possible – almost never make them public. In addition, the only *methods* that should be public are those that will definitely need to be called from outside the package.

Table 3: Effect of the access control keywords

If you specify this keyword before the definition of a method, instance variable or class variable	Then code in the following places can call the method, or read and write the variable
`public`	Any code anywhere.
`protected`	Only code in the same package as this class as well as code in any subclasses, even if they are not in the same package.
(nothing)	Only code in the same package as this class. This is the default.
`private`	Only code in this class.

Note that the keyword **`public`** is also applied to the declaration of classes. Unless a class is declared **`public`**, it is not available outside its package.

The access control keywords described above help define the notion of *scope* in Java. In general, the scope of a variable or method is the parts of the source code in which reference to the variable or method can be made. The scope of an instance variable, instance method, class variable or class method is defined by the **`public`**, **`private`** and **`protected`** keywords. Variables defined in blocks (defined by curly brackets '{' and '}'), including methods, **`try-catch`** clauses, etc. have the start and end of the block as their scope.

Threads and concurrency in Java

A **`Thread`** is sequence of executing statements that can be running, conceptually at least, *at the same time* as another **`Thread`**. Several threads are said to execute *concurrently.*

Threads are supported by many operating systems and programming languages, and their effects are visible in many application programs. For example, when you are using a word processor, you may notice that at the same time as you are typing text, the system is recalculating where to place the end of the pages. In a complex spreadsheet, at the same time as you are editing cells, the program is calculating other cells that have been affected by earlier changes you made.

To create a thread in Java, do the following:

1. Create a class to contain the code that will control the thread. This is made to implement the interface **`Runnable`**. (It can also be make a subclass of **`Thread`** instead, but implementing **`Runnable`** is by far the preferred approach.)
2. Write a method called **`run`** in your class. The **`run`** method should normally take a reasonably long time to execute – perhaps it waits for input in a loop. When the thread is started, this run method executes, concurrently with any other thread in the system. When the run method ends, the thread terminates. The run method acts like the 'main program' of the thread.
3. Create an instance of **`Thread`** (or your subclass of **`Thread`**) and invoke the **`start`** operation on this instance. If you implemented the interface **`Runnable`**, you have to pass an instance of your class to the constructor of **`Thread`**, as is shown in the example below.

The third step above will differ slightly depending on what you did in step 1. If you implemented the **`Runnable`** interface, then pass your instance to a new instance of **`Thread`**. If you directly subclassed **`Thread`**, you can simply start your instance.

The following is a complete program that illustrates how threads work. The program starts three concurrent threads, each of which prints out a record of its progress. Since the threads are running concurrently, the output from the different threads is interleaved.

Take note of the following lines of code:

- **Line 1**: This class implements **`Runnable`**.
- **Line 18**: The run method is used; it contains a loop.
- **Line 26**: The loop contains a call to **`sleep`** which delays its execution, otherwise it might finish its computation even before, any other threads have a chance to start.

- **Lines 36-38**: The three threads (instances of this class contained in instances of **`Thread`**) are created. This is done in the **`main`** method, which is the method that executes first when you execute a class.
- **Lines 40-42**: The three threads are actually started.

```
 1 public class ThreadExample implements Runnable
 2 {
 3   private int counterNumber;  // Identifies the thread
 4   private int counter;  // How far the thread has executed
 5   private int limit;    // Where counter will stop
 6   private long delay;   // Pause in execution of thread in milisecs
 7
 8   // Constructor
 9   private ThreadExample(int countTo, int number, long delay)
10   {
11     counter = 0;
12     limit = countTo;
13     counterNumber = number;
14     this.delay = delay;
15   }
16
17   //The run method; when this finishes, the thread teminates
18   public void run()
19   {
20     try
21     {
22       while (counter <= limit)
23       {
24         System.out.println("Counter "
25           + counterNumber + " is now at " + counter++);
26         Thread.sleep(delay);
27       }
28     }
29     catch(InterruptedException e) {}
30   }
31
32   // The main method: Executed when the program is started
33   public static void main(String[] args)
34   {
35     //Create 3 threads and run them
36     Thread firstThread = new Thread(new ThreadExample(5, 1, 66));
37     Thread secondThread = new Thread(new ThreadExample(5, 2, 45));
38     Thread thirdThread = new Thread(new ThreadExample(5, 3, 80));
39
40     firstThread.start();
41     secondThread.start();
42     thirdThread.start();
43   }
44 }
```

♦ **Exercise**

E 8 Run the **`ThreadExample`** class, which is available on the book's web site and observe its behaviour

E 9 If there is any aspect of the code in the above example that you do not understand, then look it up in the Java documentation.

Synchronization

In concurrent programs, a difficulty arises when two threads can both modify the same object. For example, it would be a problem if one thread were trying to remove an element from a list at the same time as another thread was accessing that element.

Java offers a special mechanism called *synchronization* to avoid such conflicts. By adding the keyword **`synchronized`** in front of the declaration of certain methods, you guarantee that only one thread at a time can run any of the synchronized methods on a given object. When a synchronized method is invoked it obtains a lock on the object; if another thread also invokes a synchronized method on the same object then it will be put in a queue, waiting for the first object to finish executing its synchronized method.

You will see several examples of synchronization in the code discussed in the next chapter.

Concluding comments about Java

In the above overview we have covered the most important features of Java, but have only scratched the surface in terms of the total number of classes and methods available. You should now have enough knowledge to work through the book, as long as you are prepared to refer to the on-line Java documentation. As we introduce certain specific topics, we will introduce more features of Java. In particular:

- In Chapter 3, as we discuss the client-server architecture, we will discuss some of Java's capabilities for communicating over the Internet.
- In Chapter 5, as we discuss class diagrams, we will show in detail how to implement such diagrams.
- In Chapter 7, as we discuss user interfaces, we will give you some tips describing how to implement a graphical user interface in Java.