OXFORD
HIGHER EDUCATION

Programming in Java

Second Edition

SACHIN MALHOTRA
SAURABH CHOUDHARY

Note If we try to represent the CPU of a computer in OOP terminology, then CPU is the object. The CPU is responsible for fetching the instructions and executing them. So fetching and executing are two possible functions (methods or behavior) of CPU. The place (attributes) where CPU stores the retrieved instructions, values and result of the execution (registers) will then be the attributes of the CPU.

1.3 PRINCIPLES OF OBJECT-ORIENTED LANGUAGES

OOP languages follow certain principles such as class, object, and abstraction. These principles map very closely to the real world.

1.3.1 Classes

A class is defined as the blueprint for an object. It serves as a plan or a template. The description of a number of similar objects is also called a *class*. An object is not created by just defining a class. It has to be created explicitly. Classes are logical in nature. For example, furniture does not have any existence but tables and chairs do exist. A class is also defined as a new data type, a user-defined type which contains two things: data members and methods.

1.3.2 Objects

Objects are defined as the instances of a class, e.g. table, chair are all instances of the class Furniture. Objects of a class will have same attributes and behavior which are defined in that class. The only difference between objects would be the value of attributes, which may vary. Objects (in real life as well as programming) can be physical, conceptual, or software. Objects have unique identity, state, and behavior. There may be several types of objects:

- *Creator objects*: Humans, Employees, Students, Animal
- *Physical objects*: Car, Bus, Plane
- *Objects in computer system*: Monitor, Keyboard, Mouse, CPU, Memory

1.3.3 Abstraction

Can you classify the following items?

- Elephant
- Television
- Table
- CD player
- Chair
- Tiger

How many classes do you identify here? The obvious answer anybody would give is three, i.e., Animal, Furniture, and Electronic items. But how do you come to this conclusion? Well, we grouped similar items like Elephant and Tiger and focused on the generic characteristics rather than specific characteristics. This is called *abstraction*. Everything in this world can be classified as living or non-living and that would be the highest level of abstraction.

Another well-known analogy for abstraction is a car. We drive cars without knowing the internal details about how the engine works and how the car stops on applying brakes. We are happy with the abstraction provided to us, e.g., brakes, steering, etc. and we interact with them. In real life, human beings manage complexity by abstracting details away. In programming, we manage complexity by concentrating only on the essential characteristics and suppressing implementation details.

1.3.4 Inheritance

Inheritance is the way to adopt the characteristics of one class into another class. Here we have two types of classes: *base class* and *subclass*. There exists a parent–child relationship among the classes. When a class inherits another class, it has all the properties of the base class and it adds some new properties of its own. We can categorize vehicles into car, bus, scooter, ships, planes, etc. The class of animals can be divided into mammals, amphibians, birds, and so on.

The principle of dividing a class into subclass is that each subclass shares common characteristics with the class from where they are inherited or derived. Cars, scooters, planes, and ships all have an engine and a speedometer. These are the characteristics of vehicles. Each subclass has its own characteristic feature, e.g., motorcycles have disk braking system, while planes have hydraulic braking system. A car can run only on the surface, while a plane can fly in air and a ship sails over water (see Fig. 1.1).

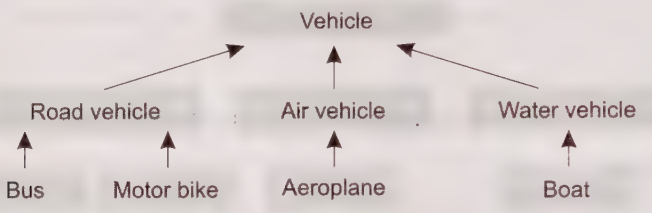


Fig. 1.1 Inheritance

Inheritance aids in *reusability*. When we create a class, it can be distributed to other programmers which they can use in their programs. This is called *reusability*. Suppose someone wants to make a program for a calculator, he can use a predefined class for arithmetic operations, and then he need not define all the methods for these operations. This is similar to using library functions in procedural language. In OOP, this can be done using the inheritance feature. A programmer can use a base class with or without modifying it. He can derive a child class from a parent class and then add some additional features to his class.

1.3.5 Encapsulation

Encapsulation is one of the features of object-oriented methodology. The process of binding the data procedures into objects to hide them from the outside world is called *encapsulation* (see Fig. 1.2). It provides us the power to restrict anyone from directly altering the data. Encapsulation is also known as *data hiding*. An access to the data has to be through the methods of the class. The data is hidden from the outside world and as a result, it is protected. The details that are not useful for other objects should be hidden from them. This is called *encapsulation*. For example, an object that does the calculation must provide an interface to obtain the result. However, the internal coding used to calculate need not be made available to the requesting object.

Fig. 1.2 Diagrammatic Illustration of a Class to Show Encapsulation

1.3.6 Polymorphism

Polymorphism simply means many forms. It can be defined as the same thing being used in different forms. For example, there are certain bacteria that exhibit in more than one morphological form. In programming, polymorphism is of two types: *compile-time* and *runtime polymorphism*. Runtime polymorphism, also known as *dynamic binding* or *late binding*, is used to determine which method to invoke at runtime. The binding of method call to its method is done at runtime and hence the term *late binding* is used. In case of compile-time polymorphism, the compiler determines which method (from all the overloaded methods) will be executed. The binding of method call to the method is done at compile time. So the decision is made early and hence the term *early binding*. Compile-time polymorphism in Java is implemented by *overloading* and runtime polymorphism by *overriding*. In overloading, a method has the same name with different signatures. (A *signature* is the list of formal argument that is passed to the method.) In overriding, a method is defined in subclass with the same name and same signature as that of parent class. This distinction between compile-time and runtime polymorphism is of *method invocation*. Compile-time polymorphism is also implemented by operator overloading which is a feature present in C++ but not in Java. Operator overloading allows the user to define new meanings for that operator so that it can be used in different ways. The operator (+) in Java is however an exception as it can be used for addition of two integers as well as concatenation of two strings or an integer with a string. This operator is overloaded by the language itself and the Java programmer cannot overload any operator.

1.4 PROCEDURAL LANGUAGE VS OOP

Table 1.1 highlights some of the major differences between procedural and object-oriented programming languages.

Table 1.1 Procedural Language vs OOP

Procedural Language	OOP
• Separate data from functions that operate on them.	• Encapsulate data and methods in a class.
• Not suitable for defining abstract types.	• Suitable for defining abstract types.
• Debugging is difficult.	• Debugging is easier.
• Difficult to implement change.	• Easier to manage and implement change.
• Not suitable for larger programs and applications.	• Suitable for larger programs and applications.
• Analysis and design not so easy.	• Analysis and design made easier.
• Faster.	• Slower.
• Less flexible.	• Highly flexible.
• Data and procedure based.	• Object oriented.
• Less reusable.	• More reusable.
• Only data and procedures are there.	• Inheritance, encapsulation, and polymorphism are the key features.
• Use top-down approach.	• Use bottom-up approach.
• Only a function call another.	• Object communication is there.
• Example: C, Basic, FORTRAN.	• Example: JAVA, C++, VB.NET, C#.NET.

Fig. 2.1 Java Applets Running on the Client System

Still Java was not popular for the client-side because of the following reasons:

- **Less Impressive GUI** Java's early GUI (AWT) was primitive. The newer GUI (Swing) was not shipped until the late 90's (and Swing is still not supported by most modern browsers without plug-ins).
- **Microsoft's Strong Presence** Nearly 95% of the desktop world uses Microsoft.
- **Clients' Software Upgradation** Good alternative methods were found to update clients' software automatically (without having to download Java on-the-fly application code each time).
- **Success of DHTML** Browsers have their own dynamic capabilities and many developers found it easier to code in DHTML. In addition, DHTML pages tend to download and start faster than Java applets.

Figure 2.2 shows how Java could be used as middle-tier services between the database and a client browser. In 1997, Sun developed servlets, so that Java could be used to generate dynamic content based on clients' request. In 1999, Sun released its Java 2 Enterprise Edition (J2EE).

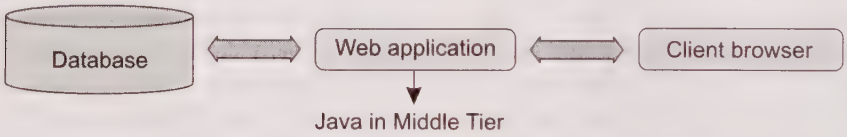


Fig. 2.2 Middle-tier Capabilities of Java to Run in Web/Application Server

Enterprise Java described how to build middle-tier components. Sun defined Enterprise Java Beans for developing business logic. The J2EE framework allows developers to concentrate on building applications rather than mulling over scalability, reliability, and security issues which are handled by the Web/application server vendors.

2.4 JAVA ESSENTIALS

Java is a platform-independent, object-oriented programming language. Java encompasses the following features:

- **A High-level Language** Java is a high-level language that looks very similar to C and C++ but offers many unique features of its own.
- **Java Bytecode** Bytecode in Java is an intermediate code generated by the compiler, such as Sun's `javac`, that is executed by the JVM.
- **Java Virtual Machine (JVM)** JVM acts as an interpreter for the bytecode, which takes bytecodes as input and executes it as if it was a physical process executing machine code.

Java is designed to be architecturally neutral so that it can run on multiple platforms. The same runtime code can run on any platform that supports Java. To achieve its cross-architecture capabilities, the Java compiler generates architecturally neutral bytecode instructions. These instructions are designed to be both easily interpreted on any machine and easily translated into native machine code on-the-fly, as shown in Fig. 2.3. Java Runtime Environment (JRE) includes JVM, class libraries, and other supporting files.

JRE = JVM + Core Java API libraries

JDK = JRE + development tools like compilers

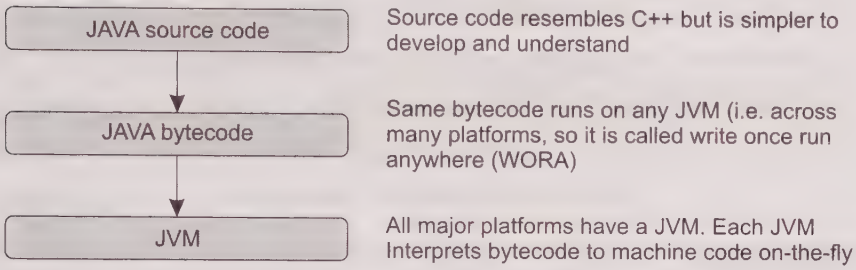


Fig. 2.3 Java Runtime Environment

Tools such as `javac` (compiler), java (interpreter), and others are provided in a bundle, popularly known as Java Development Kit (JDK). JDK comes in many versions (enhanced in each version) and is different for different platforms such as Windows and Linux. A runtime bundle is also provided as a part of JDK (popularly known as Java Runtime Environment).

2.5 JAVA VIRTUAL MACHINE

At the heart of the Java platform lies the JVM. Most programming languages compile the source code directly into machine code, suitable for execution on a particular microprocessor architecture. The difference with Java is that it uses bytecode, an intermediate code.

Java bytecode executes on a virtual machine. Actually, there wasn't a hardware implementation of this microprocessor available when Java was first released. Instead, the processor architecture is emulated by software known as the *virtual machine*. This virtual machine is an emulation of a real Java processor—a machine within a machine (Fig. 2.4). The virtual machine runs on top of the operating system, which is demonstrated in Fig. 2.5.

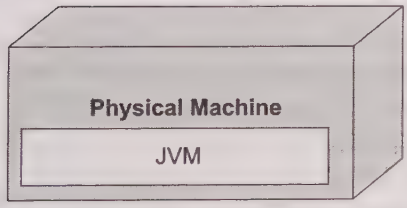


Fig. 2.4 JVM Emulation Run on a Physical CPU

The JVM is responsible for interpreting Java bytecode, and translating this into actions or operating system calls. The JVM is responsible for catering to the differences between different platforms and architectures in a way that the developers need not be bothered about it.

The JVM forms a part of a large system, the JRE. JRE varies according to the underlying operating system and computer architecture. If JRE for a given environment is not available, it is impossible to run the Java software.

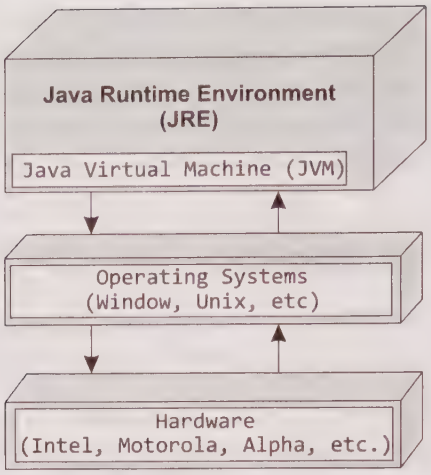


Fig. 2.5 JVM Handles Translations

2.6 JAVA FEATURES

Here we list the basic features that make Java a powerful, object-oriented, and popular programming language.

2.6.1 Platform Independence

Java was designed not only to be cross-platform in source form, like the previous languages (C, C++), but also in compiled binary form. To achieve this, Java is compiled to an intermediate form called the *bytecode* (see Figs 2.3 and 2.4). This *bytecode* is a platform-independent code that is read by a special native program called the Java interpreter that executes the corresponding native machine instructions. The Java compiler is also written in Java. The bytecodes are precisely defined to remain uniform on all platforms.

The second important part of making Java cross-platform is the uniform definition of architecture-dependent constructs. In contradiction to other languages, integers in Java are always four bytes long, and floating point variables follow the IEEE floating point arithmetic 754 standard. You don't have to worry about the meaning of any type, as it is not going to change when you transit between different architectures, e.g., Pentium to Sparc. In Java, everything is well defined. However, the virtual machine and some of its parts have to be written in native code, thus making it platform-dependent.

2.6.2 Object Oriented

It is conceived that Java is a *pure* object-oriented language, meaning that the outermost level of data structure in Java is the *object*. Everything in Java (constants, variables, and methods) are defined inside a class and accessed through objects. Java has been developed in a way that it allows the user to not only learn object-oriented programming but to apply and practise it.

But there are some constraints that violate the purity of Java. It was designed mainly for OOP, but with some procedural elements. For example, Java supports primitive data types that are not objects.

2.6.3 Both Compiled and Interpreted

Java incorporates the elements of both interpretation and compilation. Here is more information on these two approaches.

Interpretation

An interpreter reads one line of a program and executes it before going to the next line. The line is parsed to its smallest operations, the corresponding machine-level code is found, and then the instruction is executed (this could be done with something like the switch statement in C with every possible operation-case listed). Basic was one of the earliest interpreted languages where each text line was interpreted. Similarly, scripting languages like JavaScript, VBScript, and PHP are also interpreted.

In interpretation, there are no intermediate steps between writing/modifying the code and running it. The best part is: *debugging* is fast. Also, the programs are easily transportable to other platforms (if an interpreter is available). The drawback is its slow performance.

Compilation

The program text file is first converted to native machine code with a program called a *compiler*. A linker may also be required to connect together multiple code files together. The output of the compiler is an executable code. C and C++ are both compiled languages.

The biggest advantage of a compiled language is its fast performance, since the machine language code instructions load directly into the processor and get executed. In addition, the compiler can perform certain optimization operations because it looks at the program as a whole and not line by line. The disadvantages include slower debugging and reduced portability to other platforms. The source code must be recompiled on the destination platform.

Java Approach

Java incorporates both interpretation and compilation. The text program is compiled to the intermediate code, called *bytecode*, for the JVM. The JVM executes the bytecode instructions. In other words, JVM interprets the bytecode. The bytecode can run on any platform on which a JVM has been deployed. The program runs inside the JVM, so it does not bother which platform it is getting executed on.

Thus, Java offers the best of both worlds. The compilation step allows for code optimization and the JVM makes way for portability. Figure 2.4 will give you an idea about the two phases involved in the execution of a Java source program, i.e., *compile time* and *execution time* (*runtime*).

Once the source code is converted to bytecode or class file, it is loaded so that it can be processed by the execution engine of the JVM. Bytecode is loaded either through the bootstrap class loader (sometimes referred to as the *primordial class loader*) or through a user-defined class loader (sometimes referred to as the *custom class loader*). The bootstrap class loader (part of the JVM) is responsible for loading trusted classes (e.g., basic Java class library classes). User-defined class loaders (not part of JVM) are the subclasses of *java.util.Class Loader* class that are compiled and instantiated just like any other Java class. The *bytecode verifier* verifies the code and ensures that the code is fit to be executed by the JVM. Figure 2.6 shows the flow of data and control from Java source code through the Java compiler to the JVM. The code is not allowed to execute until it has passed the verifier's test.

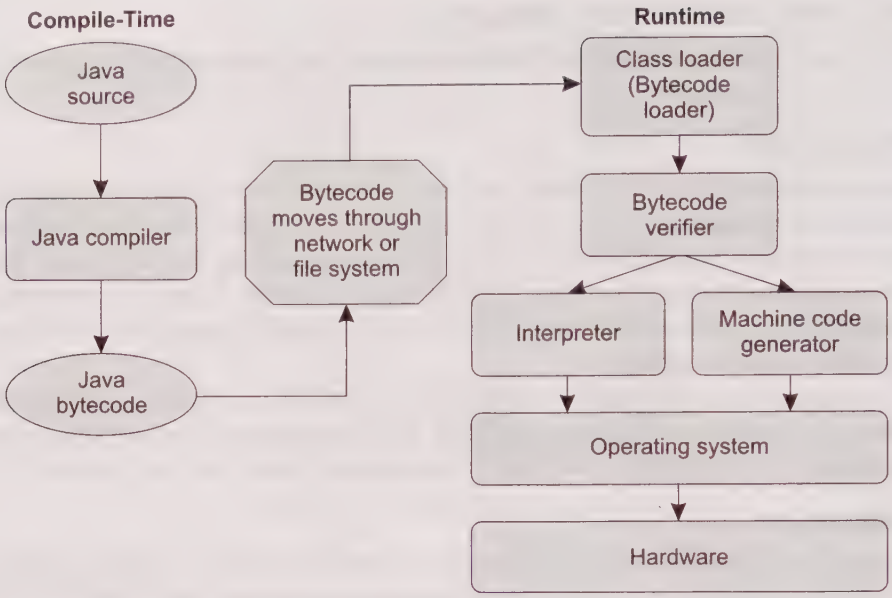


Fig. 2.6 Compilation and Interpretation in Java

But there remains the drawback of an extra compilation step after every correction during debugging. Also, the interpretation of bytecode is still slower in many cases than a program in local machine code. Advanced JVM can ameliorate this, and in many cases, reach speeds similar to programs compiled to local machine code.

2.6.4 Java is Robust

The type checking of Java is at least as strong as that of C++. The compile-time and runtime checks in Java catch many errors and make them crash-proof. The program cannot crash the system. To sum up, Java is one of the most robust languages to have ever evolved. Automatic garbage collection of allocated memory is the biggest contributor here.

2.6.5 JAVA Language Security Features

Java has several language features that protect the integrity of the security system and prevent several common attacks.

Security Through Definition Java is strict in its definition of the language:

- All primitive data types in the language have a specific size.
- All operations are defined to be performed in a specific order.

Security Through Lack of Pointer Arithmetic Java does not have pointer arithmetic, so Java programmers cannot forge a pointer to memory. All methods and instance variables are referred to with their symbolic names. Users cannot write a code that interprets system variables or accesses private information stored in a system.

Security Through Garbage Collection Garbage collection makes Java programs more secure and robust by automatically freeing memory, once it is no longer needed.

Security Through Strict Compile-Time Checking The Java compiler performs extensive, stringent compile-time checking so that as many errors as possible can be detected by the compiler. The Java language is strongly typed, that is:

- Objects cannot be cast to a subclass without an explicit runtime check.
- References to methods and variables of a class are checked to ensure that the objects are of the same class.
- Primitives and objects are not interconvertible.

Strict compilation checks make Java programs more robust and avoid runtime errors. The bytecode verifier runs the bytecode generated by the compiler when an applet is loaded and makes security checks. The compiler also ensures that a program does not access any uninitialized variables.

Java Security Model

Java's security model is focused on protecting users from hostile programs downloaded from untrusted sources across a network. Programs downloaded over the Internet are executed in a *sandbox*. It cannot take any action outside the boundaries specified by the sandbox.

The sandbox for untrusted Java applets, for example, prohibits many activities, including

- Reading or writing to the local disk
- Making a network connection to any host, except the host from which the applet came
- Creating a new process
- Loading a new dynamic library and directly calling a native method

By making it impossible for the downloaded code to perform certain actions, Java's security model protects the user from the threat of hostile codes.

Sandbox—Definition

Traditionally, you had to trust a software before you ran it. You achieved security by allowing a software from trusted sources only, and by regularly scanning for viruses. Once a software gets access to your system, it has full control and if it is malicious, it can damage your system because there are no restrictions placed on the software by the computer. So, in the first place, you prevent malicious code from ever gaining access to your system.

The sandbox security model makes it easier to work with the software that comes from untrusted sources by restricting codes from untrusted sources from taking any actions that could possibly harm your system. The advantage is—you don't need to figure out what code is trusted and what is not. In addition to that, you don't need to scan for viruses as well. The sandbox is made up of the following components operating together.

Class Loader It is the first link in the security chain. It fetches executable codes from the network and enforces the namespace hierarchy.

Bytecode Verifier The verifier checks that the applet conforms to the Java language guarantees and that there are no violations like stack overflows, namespace violations, illegal data type casts, etc.

Security Manager It enforces the boundary of the sandbox. Whenever an applet performs an action which is a potential violation, the security manager decides whether it is approved or not.

2.6.6 Java is Multithreaded

To explore this property, you must know the meaning of *multithreading*.It can be explained well with the help of an example. Consider a four-gas burner on which food is cooked. The cook, in order to save time, puts milk to boil on one gas burner, rice on the other, makes *chapattis* on the third, and vegetable on the fourth. The cook switches between all the items to be cooked so that neither of the items are red-heated to lose their taste. He may lower/brighten up the gas as and when required. Here the cook is the processor and the four items being cooked are threads. The processor (cook) switches from one thread to another.

A *thread* can be loosely defined as a separate stream of execution that takes place simultaneously and independent of everything else that might be happening. Threads are independent parts of a process that run concurrently. Using threads, a program cannot hold the CPU for a long duration intentionally (e.g. infinite loop). The beauty of multithreading is that the other tasks that are not stuck in the loop can continue processing without having to wait for the stuck task to finish. Threads in Java can place locks on shared resources so that while one thread is using it, no other thread is allowed to access it. This is achieved with the help of *synchronization*.

More about threads and its implementation will be taken up later in Chapter 8.

2.6.7 Other Features

Automatic Memory Management

Automatic garbage collection (memory management) is handled by the JVM. To create an instance of a class, the 'new' operator is used (refer to Chapter 4). However, Java automatically removes objects that are not being referenced. This is known as *garbage collection*. The advantages and disadvantages of garbage collection are listed below.

Advantages

- Reduces the possibility of memory leaks, since memory is freed as needed. A memory leak occurs when the memory allocated is not released, resulting in an unnecessary consumption of all the available memory.
- Memory corruption does not occur.

Disadvantage

- Garbage collection is considered one of the greatest bottlenecks in the speed of execution.

Dynamic Binding

The linking of data and methods to where they are located is done at runtime. New classes can be loaded at runtime. Linking is done on-the-fly, i.e., on-demand.

Good Performance

Interpretation of byte code slowed performance in early versions, but advanced virtual machines with adaptive optimization and just-in-time compilation (combined with other features) provide high speed code execution.

Built-in Networking

Java was designed with networking in mind and comes with many classes to develop sophisticated Internet communications. A detailed discussion on this topic is taken up later in Chapter 11.

No Pointers

Java uses *references* instead of pointers. A reference provides access to objects. The programmer is relieved from the overhead of pointer manipulation.

No Global Variables

In Java, the global namespace is the class hierarchy and so, one cannot create a variable outside the class. It is extremely difficult to ensure that a global variable is manipulated in a consistent manner. Java allows a modified type of the global variable called *static variable*.

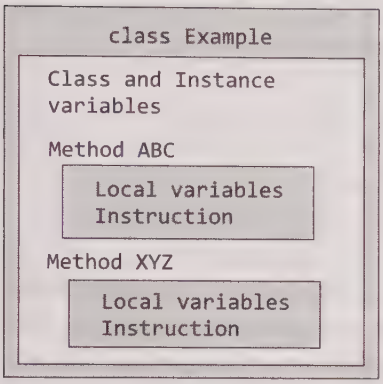


Fig. 2.7 Program Structure

2.7 PROGRAM STRUCTURE

A Java application consists of a collection of classes. A *class* is a template. An *object* is defined as an instance of the class. Each instance (object) contains the members (fields and methods) specified in the class. A *field* is one that holds a value. A *method* defines operations on the fields and values that are passed as arguments to the method (see Fig. 2.7).

Let us now create our first Java program. Example 2.1 below shows a very simple Java program which displays a string on the console. It has just one print statement (the program is explained in Section 2.7.3).

Example 2.1 **First Java Program**

```
L1      /* Call this file"Example.java".*/
L2      class Example {
L3      //your program starts execution with a call to main()
L4      public static void main (String args[ ]){
L5      System.out.println("This is a simple Java program");
L6      }
L7      }
```

2.7.1 How to Execute a Java Program?

There are three easy steps for successfully executing the Java program:

1. **Entering the Source Code** The above program (Example 2.1) can be written in any text editor (like Notepad) but make sure it is written exactly the same way it is shown.
2. **Saving the Source Code** Now that you've written the code in Notepad, this is how you'll save it
   - Select File | Save As from the notepad menu.
   - In the 'File name' field, type "Example.java" within double quotes.
   - In the 'Save as type' field select All Files (*.*).
   - Click enter to save the file.

3. **Compiling and Running the Source** Java programs are compiled using DOS. For opening OS, type *cmd* at the run prompt and move to the folder that contains the saved `Example.java` file. Now compile the program using `javac`, specifying the name of the source file on the command line as shown below. (Assuming the file was saved in a folder 'javaeg' in the C drive.)

```
C:\>cd javaeg // change to directory javaeg using cd command
C:\javaeg\>javac Example.java
```

The `javac` compiler creates a file called `Example.class` (in the same directory). This class contains the bytecode version of the program. This bytecode will be executed by the Java interpreter using `java` followed by the class name as shown below.

```
C:\javaeg\>java Example
```

Output

```
This is a simple Java program
```

2.7.2 Why Save as `Example.java`?

When the Java source code is compiled, each individual class is put in its own output file named after the class and using the `.class` extension. That is why it is a good idea to give the Java source files the same name as that of the class they contain. The name of the `.class` file will match exactly with the name of the source file.

In many programming languages, the name of the source code file can be arbitrary. This is not so with Java. In the above example, the name of the source file should be `Example.java`. In Java, a source file is a normal text file that contains one or more class definitions.

The extension for the source file must be `.java`. By convention, the name of the file and the name of class should be same (even the case should match) and that is why we named the above example as `Example.java`. Java is case-sensitive. So example and Example are two different class names.

Note You can also provide a different name for naming a source file. For example, the above example can be saved as `First.java`. But in that case, when you compile the file, the `.class` that will be generated will have the name `Example.class`. So for executing the program, you have to mention `java Example` on the command line. This may lead to confusion, so it is advised that the name of the Java file should match with the name of the class defined in the file (case-wise also). Also note that in case the source file contains more than one classes defined within itself, the java file name should match exactly with the class name that contains the main method.

2.7.3 Explanation

L1 The program begins with the comment:

```
/* Call this file "Example.java".*/
```

The comments are ignored by the compiler. Comments are a good way to induce documentation in programming.

L2 The next line of code in the program is

```
class Example {
```

This line uses the keyword `class` to declare that a new class is being defined followed by the class name, i.e., **Example**. The entire class definition, including all its members, will be between the opening curly brace ({) and the closing curly brace (}).

L3 Another type of comment is used in this line.

```
// your program starts execution with a call to main()
```

This type of comment is called a single-line comment, and it begins with a double slash //.

L4 This line shows the main method for the class.

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

This is the line from where the program will start its execution. All applications in Java start execution from `main()`. Every complete Java Application must contain a root class where the execution can begin. A root class must contain a main method defined with the header, as shown in this line. Let us take a brief look at the attributes of `main()`.

`public` It is an access specifier used to specify that the code can be called from anywhere.`main()`is declared public because it is called by codes outside the class it is a part of. It is called by the JVM.

`static` It is declared `static` because it allows `main()`to be called without having to instantiate the class. JVM need not create an instance of the class (i.e. object of the class) for calling the `main()`method.

`void` It does not return a value. The keyword `void` simply tells the compiler that `main()`does not return anything back to the caller, i.e., JVM.

`String args[]` It holds optional command line arguments passed to the class through the `java` command line. The curly bracket at the end marks the beginning of the main method and it ends in L6.

Note The Java compiler will compile classes that do not contain a `main ()` method, but the Java interpreter has no way to run these classes.

L5 It shows a print statement. If you want to display anything on the standard output, this statement is used.

```
System.out.println ("This is a simple Java program");
```

This line prints the string `"This is a simple Java Program"` on the standard output. `System` is a predefined class. The string (mentioned in double quotes) passed to the `println` method is displayed as it is on the standard output. All statements in Java are terminated by a semicolon (;). Lines other than `println()`don't end with a semicolon because they are technically not statements.

L6 The closing curly bracket marks the closing of the `main` method.

L7 The closing curly bracket marks the closing of the class.

2.8 JAVA IMPROVEMENTS

Features of different versions of Java are discussed in the following sections.

2.8.1 Java 5.0 Features

We present a host of features in Java 5 and later discuss some of the improvements in Java 5.

Autoboxing and Unboxing

Chapter 3 explains that Java has primitive types like `int` for integers, and Chapter 4 explains classes and objects. The difference between the two types is very important. In Chapter 6, we examine the so-called *autoboxing* and *unboxing* features added to J2SE 5.0 that removes the need for explicit conversions in most cases and thus improves code readability and removes boilerplate codes and sources of errors.

Enhanced `for` Loop

Chapter 3 looks at several types of looping structures available in Java, one of which is the `for` loop (quite similar to the C/C++ `for` loop). Version 5.0 includes an enhanced `for` loop syntax that reduces code complexity and enhances readability. We introduce the enhanced `for` loop in Chapter 4 and describe the object types with which the enhanced `for` loop works.

Enumerated Types

Chapter 6 presents a feature of C/C++ that many programmers have missed in Java. An enumerated type has been added with the `enum` keyword. The new enumerated type includes all the features of C/C++ `enum` including type safety.

`StringBuilder` Class

We will be discussing this class in Chapter 6, along with the older `StringBuffer` class. It offers better performance than `StringBuffer` class.

Static Import

Release 5.0 includes a new technique for accessing Java static methods and constants in another class without the need to include the full package and class name every time they are used. (We will explain the terms `class, package, static, import`, etc. in Chapters 4 and 7). The 'static import' facility makes your code easier to write and less error-prone. We will discuss static import in more detail in Chapter 7 after discussing import in general.

Metadata

The metadata facility (*annotation*) is designed to reduce much of the boilerplate code that would be required in the earlier versions of Java. Annotations, though not a part of the program, provide information about the program to the compiler. This information can be used to detect errors and supply warnings. Annotations begin with '@'. The `javac` compiler processes some annotations and some require the annotation-processing tool, *apt*.

Formatted I/O and Varargs

In Chapter 9, we discuss how to format numerical output with Java. Version 5.0 adds the ability to produce formatted output easily in the form of a `printf()`method that behaves similar to the `printf()` function in C/C++. There is also a formatted input feature (`Scanner` class) that is described in Chapter 9. Both these features rely on '`varargs`,' which stands for *variable argument list* in which the number of parameters passed to a Java method is not known when the source is constructed (also known as *variable arity methods*) (see Chapter 4 for `varargs`).

Graphics System Improvements

Release 5.0 includes numerous bug fixes and minor tweaks to Java's graphics subsystems known as AWT and Swing, including reduced memory usage. The biggest improvement is that it is no longer necessary to call `getContentPane()` when adding Swing components.

New Concurrency Features

Chapter 8 discusses Java's multithreading support that has been present since Version 1.0. Release 5.0 enhances the multithreading features of Java. Some of these additions depend upon the generics concept, so we wait until Chapter 10 to introduce these important new capabilities.

Generics

In Chapter 10, we introduce the new generics feature, an important subject that we will cover in detail. Java is type-safe, which simply means that every variable has a well-defined type and that only compatible types can be assigned to each other. However, the use of generics adds a greater amount of compile-time safety to the Java language. The use of generics allows objects of only a specified type to be added to a collection, thereby enhancing the runtime safety and correctness of the program; otherwise a compile-time error occurs.

Other new features in J2SE 5.0 include core XML support, improvements to Unicode, improvements to Java's database connectivity package known as JDBC, and an improved, high-compression format for JAR files that can greatly reduce download times for applets and other networking applications.

Java 2 platform Standard Edition 5.0 (J2SE 5.0) dealt with improvements in the ease of development (EoD) category. The new EoD features were all about syntax shortcuts that greatly reduce the amount of code that must be entered, making coding faster and error-free. Some features enable improved compile-time type checking, thus producing fewer runtime errors. Apart from EoD category, new multithreading and concurrency features were added that provide capabilities previously unavailable. The designers of J2SE considered quality, stability, and compatibility to be the most important aspect of the new release. A lot of efforts were made to ensure compatibility with previous versions of Java. Faster JVM startup time and smaller memory footprint were important goals. These have been achieved through careful tuning of the software and the use of class data sharing. It is much easier to watch memory usage, detect and respond to a low-memory condition in Java 5.

2.8.2 Java 6 Features

Some of the major enhancements to Java 6 are given below.

Collections API

The motive was to provide bidirectional collection access. New interfaces have been added like `Deque`, `BlockingDeque`, etc. and existing classes like `Linked List`, `TreeSet`, and `TreeMap` have been modified to implement these new interfaces. A bunch of new classes have been added like `ArrayDeque`, `ConcurrentSkipListSet`, etc.

Input/Output

A new class named `Console` has been added to the `java.io` package. It contains methods to access character-based console. New methods have been added to `File` class.

Jar and Zip Enhancements

Two new compressed streams have been added.

- `java.util.zip.DeflaterInputStream`:for compressing data
- `java.util.zip.InflaterOutputStream`:for decompressing data

These classes are useful for transmitting compressed data over a network.

Enhancements Common to Java Web Start and Java Plug-in

All dialogs have been redesigned to be more user-friendly. Caching can be disabled via the Java control panel. A new support for SSL/TSL is added.

Enhanced Network Interface

It provides a number of new methods for accessing state and configuration information relating to a system's network adapters. This includes information such as MAC addresses and MTU size (discussed in Chapter 11).

Splash Screen

Applications can display the splash screen even before the virtual machine starts.

Java 6 also enhanced the monitoring and mangement API and made significant changes to JConsole.

2.8.3 Java 7 Features

A number of features have been added in Java 7 such as revised `switch...case` to accept strings, `multi-catch` statements in exception handling,`try-with-resource` statements, the new file input output API, the fork and join framework and a few others.

String in `switch...case` Statement

Java 7 added strings to be used in `switch...case` statements apart from primitives (`short, byte, int, char`), enumerated type and few wrapper classes (discussed in Chapter 3).

Unicode 6.0.0 Support

Java 7 supports Unicode 6.0.0. A new string representation is used to express unicode characters (discussed in Chapter 6).

Binary Literals and Numeric Literals (with Underscores)

Java 7 added binary literals and underscores to be used with numeric literals. This feature is particularly useful in increasing the readability of larger literals with a long sequence of numbers (discussed in Chapter 3).

Automatic Resource Management

A new try with resources statement is introduced so that resources specified with try are released/ nullified when try block exits. There is no need to manually free up the resources using finally block as was the case with earlier versions of Java (discussed in Chapter 7).

Improved Exception Handling

Java 7 introduced a multi-catch block where multiple exceptions can be caught using a single catch block (discussed in Chapter 7).

nio 2.0 (Non-blocking I/O)—New File System API

`java.nio.file` package was created in Java 7 to include classes and interfaces like `Path`,`Paths`, `File System`, `File Systems` and others. Simplified methods to efficiently copy, move, create links and receive file/directory change notifications were also incorporated (Chapter 9).

Fork and Join Fork and Join Framework is incorporated in Java 7 to have a more efficient kind of parallel processing. The task is divided (forked) into smaller task such that no thread is idle and whose results are combined (joined) to achieve the desired outputs. The classes for the Fork-Join mechanism are `ForkJoinPool` and `ForkJoinTask`.

Supporting Dynamism Java compiler performs the type checking of variables, methods, arguments etc. Java 7 incorporates a new feature `invokedynamic` to let JVM resolve type information at runtime like few other dynamic languages and incorporate non-java language requirements.

Diamond Operator The Generics declaration, prior to Java 7, required the types to be declared on both the sides of the declaration. Java 7 onwards the compiler can deduce the type on the right side, using the diamond operator (< >), by looking at the left-hand-side declaration.

Swing Enhancements Swings added a host of features like AWT and Swing components can be used together without any problems,`JLayer` class, Nimbus look and feel, HSV color selection tab in the `JColorChooser` class and more (see Chapter 15 for details).

Java FX 2.2.3 Java FX provides the new GUI toolkit for creating rich cross-platform user interfaces across different types of devices like TV, mobile, desktop etc. Java FX is bundled with JDK 7.

2.8.4 Brief Comparison of Different Releases

Table 2.1 presents a brief comparison of different releases of Java.

Table 2.1 Java JDK Major Releases and their Differences

Version	Name	New Features Introduced
1.0	Oak	Java released to public.
1.1	Sparkler	Added a totally new event model, using Listeners, anonymous classes, and inner classes.
1.2	Playground	Added `Array List` and other `Collections`, added swing. Added DSA code signing. Added buffered image.
1.3	Kestrel	`java.util.Timer`,`java.lang.StrictMath`,`java.awt.print.Page Attributes`, `java.media.sound` (MIDI) Hotspot introduced. RMI can now also use CORBA's IIOP protocol. Added RSA code signing.
1.4	Merlin	Added `regexes`, `assertions`, and `nio`.
1.5	Tiger	Added `StringBuilder`, `java.util.concurrent`, generics, enumerations and, annotations.
1.6	Mustang	`Applet` splash screens, table sorting, true double buffering, digitally signed XML files, `JavaCompilerTool`, JDBC 4.0, smart card API, `Console.readPassword`, improved drag and drop.
1.7	Dolphin	Automatic resource management, String in `switch...case`, Fork and join framework, dynamism support, Unicode 6 supported, Java Fx 2.2.3.
1.8	Not yet released	There is still on-going discussion on what should be included.

2.9 DIFFERENCES BETWEEN JAVA AND C++

Here is a technical overview of the differences between Java and C++. The following points list out the aspects that are present in Java and absent in C++.

Multiple Inheritance Not Allowed Multilevel inheritance is enforced, which makes the design clearer. Multiple inheritance among classes is not supported in Java. Interfaces are used for supporting multiple inheritance.

Common Parent All classes are *single-rooted*. The class `Object` is the parent of all the classes in Java.

Packages The concept of *packages* is used, i.e., a large, *hierarchical namespace* is provided. This prevents naming ambiguities in libraries.

In-source Documentation *In-source code documentation* comments are provided. Documentation keywords are provided, e.g. `@author`, `@version`, etc.

All Codes Inside Class Unlike C++, all parts of a Java program reside inside the class. Global data declaration outside the class is not allowed. However, `static` data within classes is supported.

Operator Overloading Operator overloading is not supported in Java but a few operators are already overloaded by Java, e.g. '+'. Programmers do not have the option of overloading operators.

Explicit boolean Type `boolean` is an explicit type, different from `int`. Only two `boolean` literals are provided, i.e. `true` and `false`. These cannot be compared with integers 0 and 1 as used in some other languages.

Array Length Accessible All array objects in Java have a `length` variable associated with them to determine the length of the array.

`go to` Instead of `goto`, `break` and `continue` are supported.

Pointers There are no pointers in Java.

Null Pointers Reasonably Caught Null pointers are caught by a `NullPointerException`.

Memory Management The use of *garbage collection* prevents memory leaks and referencing freed memory.

Automatic Variable Initialization All variables are automatically initialized, except local variables.

Runtime Checking of Container Bounds The bounds of containers (arrays, strings, etc.) are checked at runtime and an `IndexOutOfBoundsException` is thrown if necessary.

Platform Independence C++ is not a platform-independent language whereas Java is.

Sizes of the Integer Types Defined The sizes of the integer types `byte`, `short`, `int`, and `long` are defined to be 1, 2, 4, and 8 bytes.

Unicode Provided Unicode represents the characters in most of the languages, e.g. Japanese, Latin, etc.

String Class An explicit predefined `String` class is provided along with `StringBuffer` and new `StringBuilder` class.

Extended Utility Class Libraries: Package java.util Supported among others, `Enumeration` (an `Iterator` interface), `Hashtable`, `Vector`.

Default Access Specifier Added By default, all the variables, methods, and classes in Java have default privileges that are different from `private` access specifier. `Private` is the default access specifier in C++.

2.10 INSTALLATION OF JDK 1.7

Before writing a single line of code, the software application developer must first make sure that the best tool for the job are at his or her disposal. Java was designed to be a cross-platform, object-oriented programming language. Because of the huge amount of interest generated by the introduction of Java, new tools are being introduced every now and then that provide the developer with greater flexibility and ease of use.

2.10.1 Getting Started with the JDK

Sun (and now continued by Oracle) decided to give away a Java Developer's Kit (JDK) that would provide the basic tools needed for Java programming. The JDK provides the beginners with all the tools needed to write powerful Java applications or applets. It contains a compiler, an interpreter, a debugger, sample applications, applet viewer, and some other tools that you can use to test your code.

A quick visit to Oracle Java website will allow you to download the JDK to your local machine. Check for the latest version of JDK and download that from this site. The following operating systems are supported for JDK:

(a) Oracle Solaris (b) Windows (c) Linux (d) Mac

Remember that the availability of JDK for these platforms simply means that Oracle has implemented the JVM and development tools for these platforms.

2.10.2 JDK Installation Notes

When the Java SE Development Kit is installed, the Java SE Runtime Environment is installed as well.

Note For the installation of JDK 1.7 on Solaris platform (both 32-bit and 64-bit), you can refer to the installation documentation on Oracle official site:

http://docs.oracle.com/javase/7/docs/webnotes/install/solaris/solaris-jdk.html

Similarly, for installation of JDK 1.7 on Linux operating system (both 32 bit and 64 bit), visit:

http://docs.oracle.com/javase/7/docs/webnotes/install/linux/linux-jdk.html

For JDK installation on MAC OS visit:

http://docs.oracle.com/javase/7/docs/webnotes/install/mac/mac-jdk.html

The JDK for any OS can be downloaded from:

www.oracle.com/technetwork/java/javase/downloads/jdk7u9-downloads-1859576.html.

Refer to www.oracle.com/technetwork/java/javase/downloads/index.html for latest Java SE releases.

In this book, we intend to provide the details of installation of JDK 1.7 on Windows operating system only.

JDK has two versions numbers—an *external version number* 7 and an *internal version number*1.7.0_09, i.e., version 7 update 9.

The installation and configuration process can be broken down into the following steps:

1. Run the JDK installer.
2. Update the Path and Classpath variables.
3. Test the installation.

Step 1: Run the JDK Installer

If you have downloaded the JDK software file (JDK installer) instead of running the installer from the Java website, you should check to see that the complete file is downloaded:

```
jdk-7u9-windows-i586.exe
```

Note The JDK documentation can be downloaded from the following URL: www.oracle.com/technetwork/java/javase/documentation/java-se-7-doc-download-435117.html.

Double-click on the icon of the JDK Installer.exe to run the installer and then follow the instructions. Figures 2.8(a)–(h) show some of the snapshots of the installation process. The first Welcome screen is displayed as soon as you double click on the installer.

Fig. 2.8(a)

The welcome screen also tells you that Java FX SDK is now a part of Java 7. Click on `Next>`, the installer prompts you to select what all you want to install and where to install them in your system.

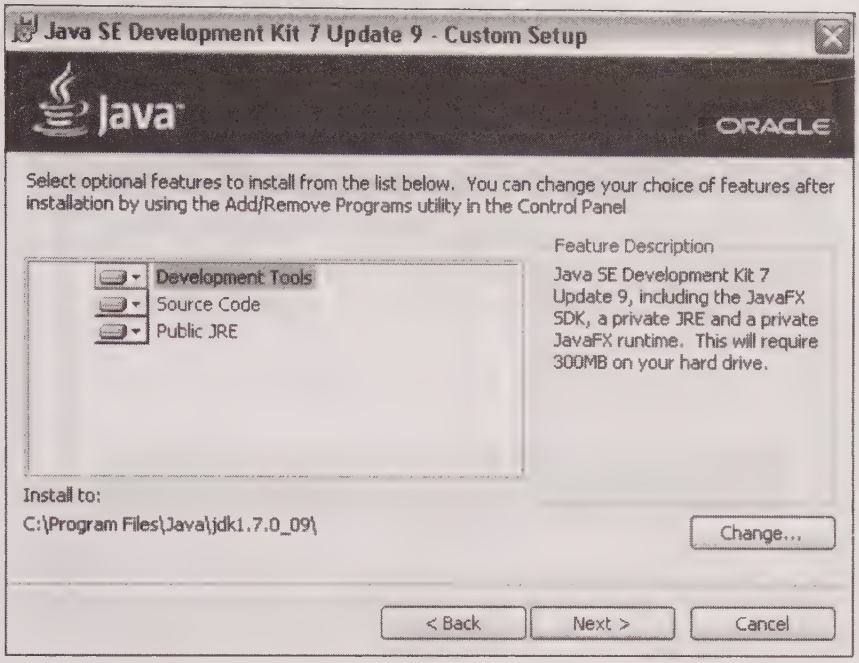


Fig. 2.8(b)

By default, the JDK will be installed at the path mentioned in `Install to`. You can change the default path by clicking the `Change...` button. As soon as you click on the `Next>` button, the installation starts.

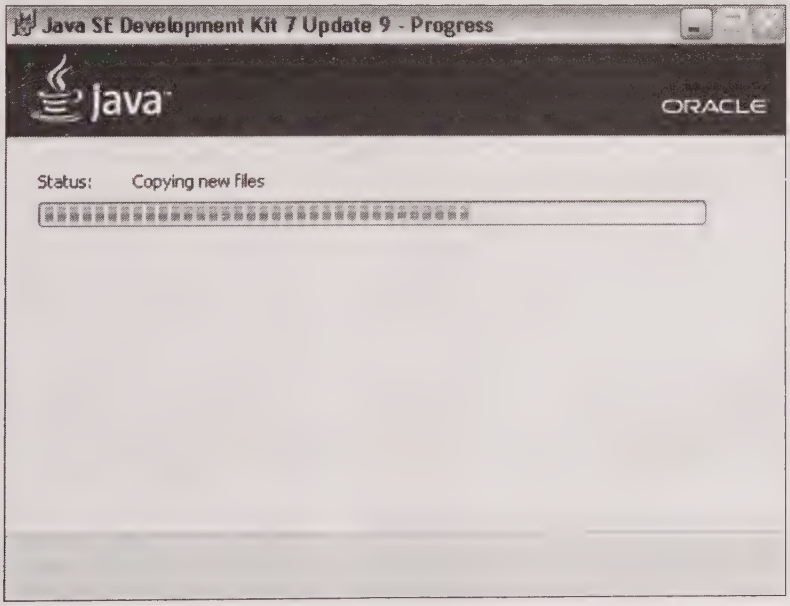


Fig. 2.8(c)

Figure 2.8(d) snapshot shows that JRE will be installed.

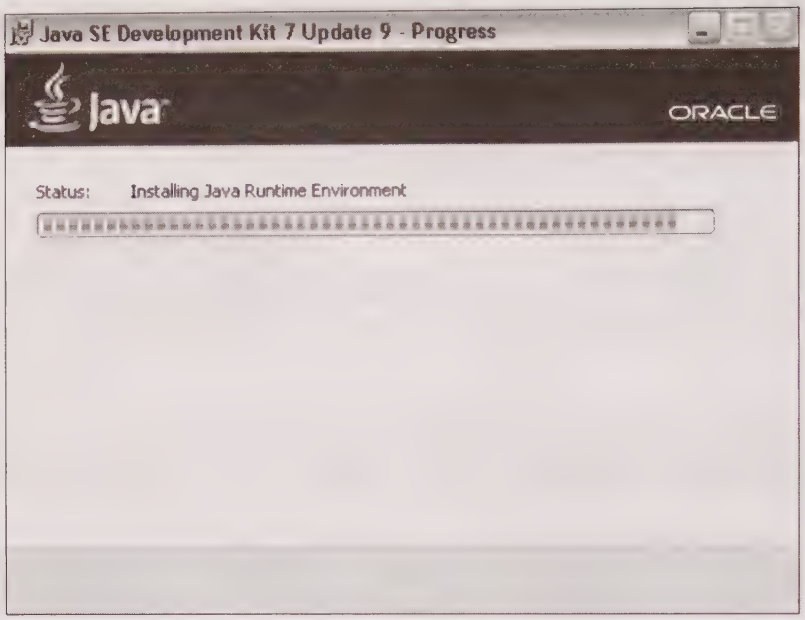


Fig. 2.8(d)

Figure 2.8(e) prompts you to make a selection for installing JRE. As soon as you click on `Next>`, the installation of JRE starts.

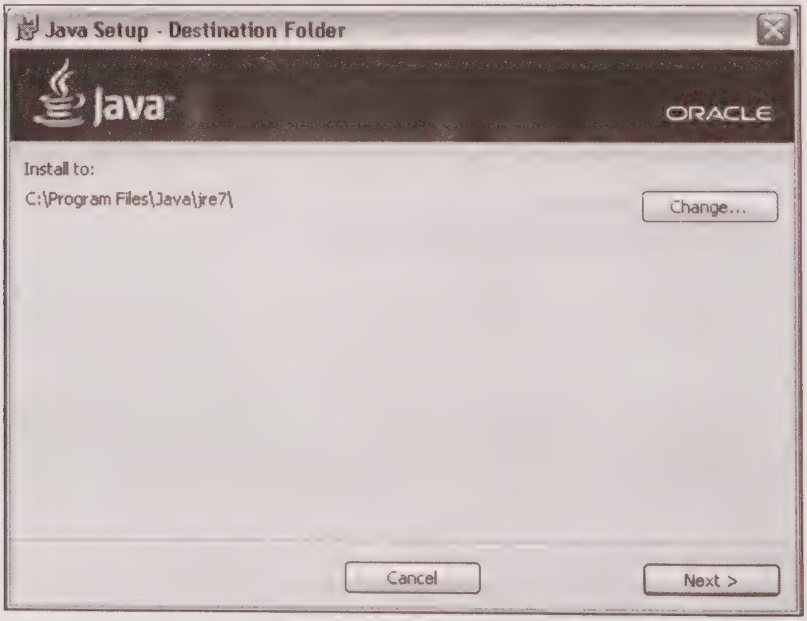


Fig. 2.8(e)

The following snapshot shows you that the JRE is getting registered.

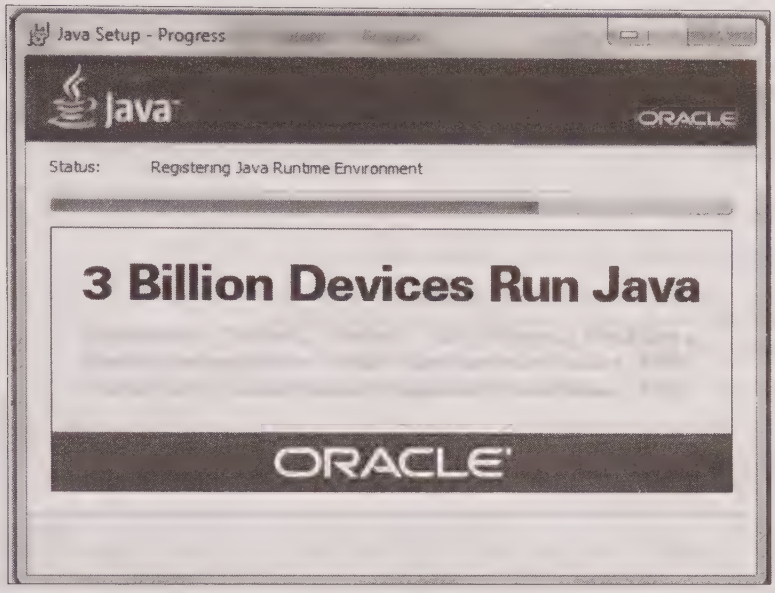


Fig. 2.8(f)

Finally, Java is installed successfully as shown in the snapshot below.

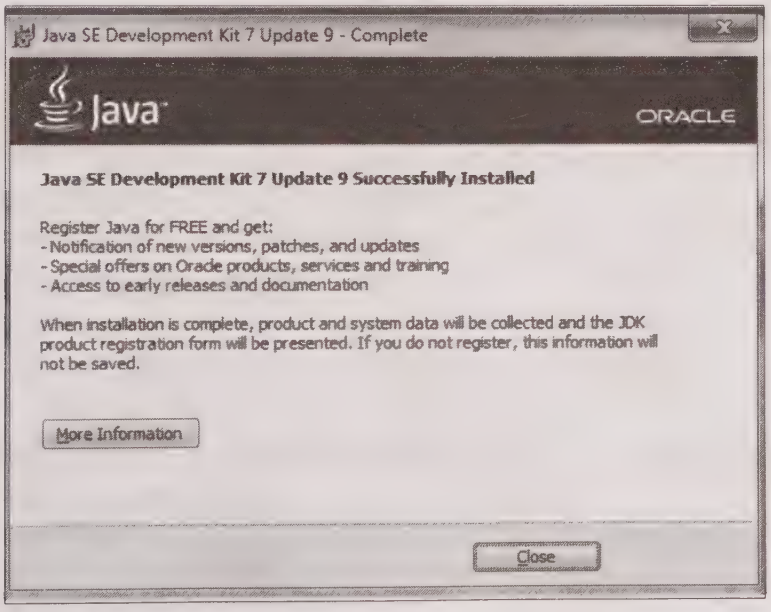


Fig. 2.8(g)

Once you are finished with installation of Java, you get a 'Thank You' message (Fig. 2.8(h)) from Oracle Corporation and asking you to register so that you can get alerts, notifications, special offers, and access to future releases and documentation.

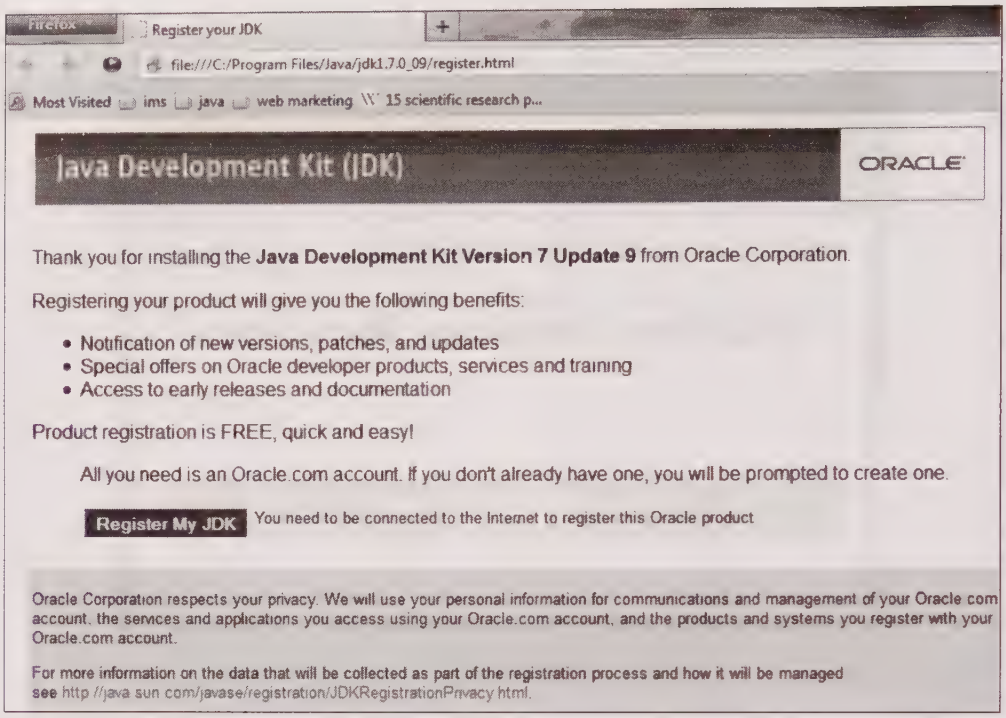


Fig. 2.8(h)

Installed Directory Structure

JDK 7 will be installed (by default) in `c:\program files\java\jdk1.7.0_09` and will have the following directory structure (Fig. 2.9):

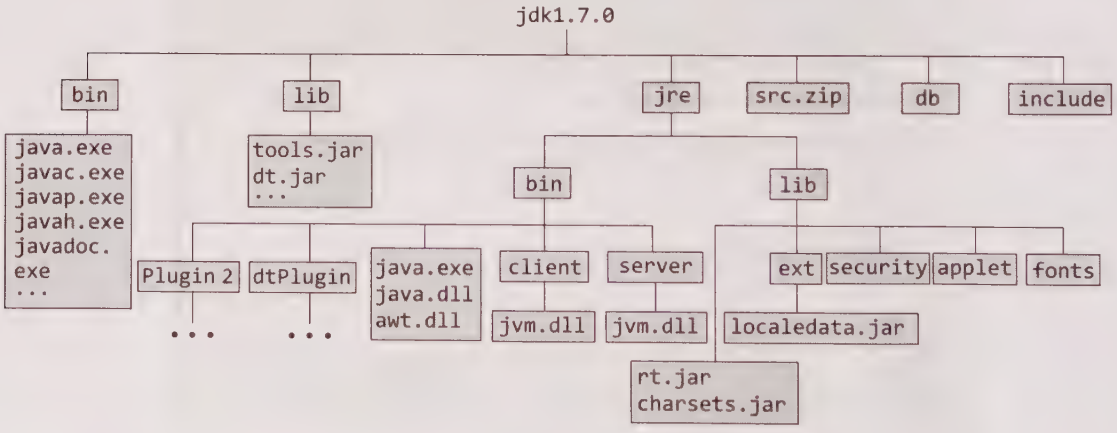


Fig. 2.9 Structure of JDK Software and Documentation Directories

Included in the directory structure is a file `src.zip`. Do not unzip the src.zip file as it contains all the core class binaries, and is used by JDK in this form.

- **include** The include directory contains a set of C and C++ header files for interacting with C and C++.
- **lib** This directory contains non-core classes like *dt.jar* and *tools.jar* used by tools and utilities in JDK.
- **bin** The bin directory contains the binary executables for Java. For example, Java Compiler (Java), Java Interpreter (Java) ,rmicompiler (rmic) etc.
- **jre** It is the root directory for the Java runtime environment.
- **db** Contains Java database.

Step 2: Update Path and Classpath Variables

It is not possible to run a Java program without modifying system environment variables (such as Path or Classpath) or modifying the *autoexec.bat.*

Why to Set Path Variable? The PATH environment variable needs to be set if you want to run the executables (javac.exe, java.exe, javadoc.exe, etc.) from any directory. If you want to find out the current value of your PATH, then type the following at the DOS prompt:

```
C:\>path
```

Windows NT/XP/Vista/7 It is preferable to make the following environment variable changes in the Control Panel instead of the autoexec.bat file. Start the Control Panel, select System, and then edit the environment variables. In case of other recent versions of Windows, right click on the My Computer icon, and select Properties, click on Environment. The System Properties window appears. Select Path from the list of system variables and append the following path to existing path: `C:\PROGRA~1\JAVA\JDK1.7\bin` (complete path of `\bin`).

Note Do not erase the existing paths in the path system variable; only append the new path separated by a semicolon.

`Classpath`—What it does? The `Classpath` tells the JVM and other Java applications where to find the class libraries and user-defined classes. You need to set the classpath for locating class libraries, user-defined classes, and packages.

Setting the Classpath

The same procedure (explained above) can be followed for setting the classpath environment variable with the exception that now you will not look for the path variable but for the classpath variable.

Step 3: Testing the Installation

Your computer system is now configured and ready to use the JDK. The Java tools do not have a GUI, as they are all run from the DOS command line. For testing the installation, type the following command at the command line:

```
C:\>javac and C:\>java
```

If the following screenshots are displayed on typing the command 'java' on the DOS prompt, it means that Java is properly installed and the path is set (Figs 2.10(a) and (b)).

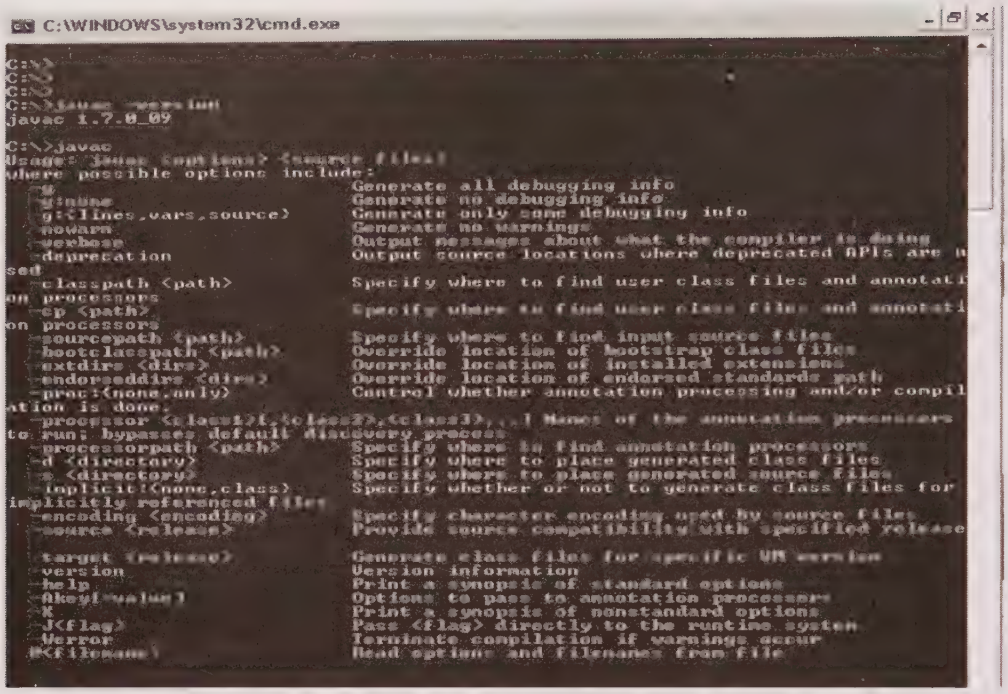


Fig. 2.10(a)

Fig. 2.10(b)

2.10.3 Exploring the JDK

It is important to know the complete structure of Java 7. The following diagrammatic representation (Fig. 2.11) can give you an idea about the various constructs Java 7 is made up of. It also shows the various components taken care by each construct.

It will be interesting to know, which part(s) of Java 7 will comprise the JDK, JRE, or Java API? The following representation answers the question in contention. It is worth noting that the Java API is a subset of JRE and the JRE is a subset of JDK.

The Java Standard Edition (Java™ SE) Development Kit includes the JRE plus the command-line development tools such as compilers and debuggers that are necessary for *developing* applets and applications.

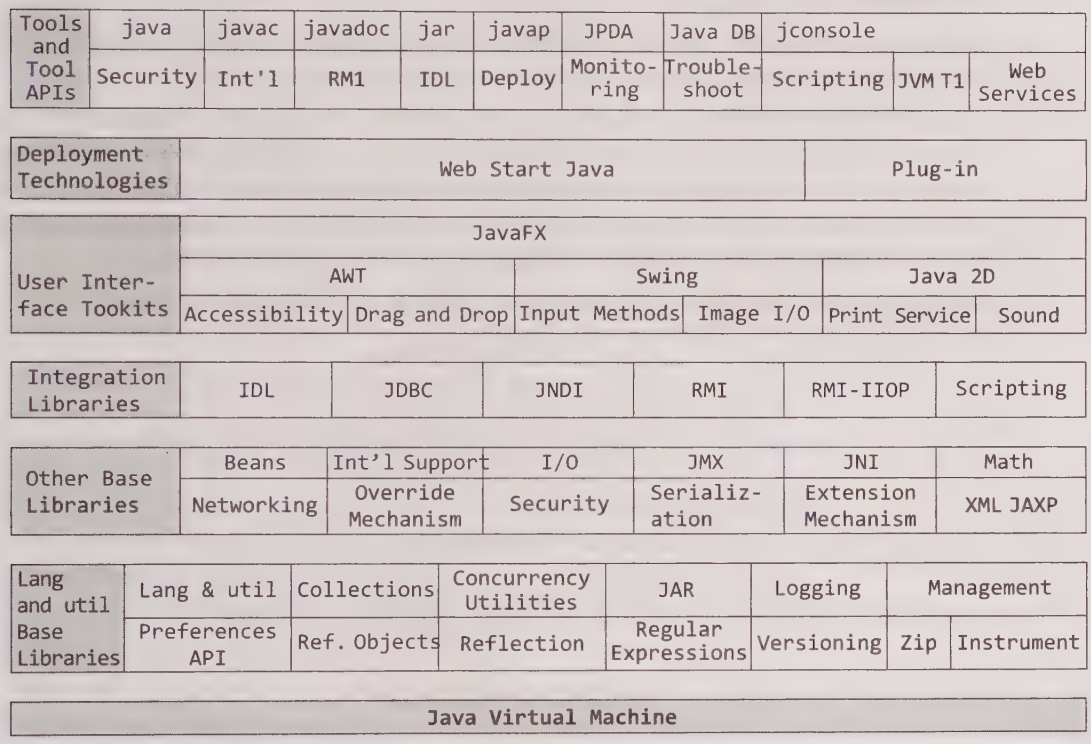


Fig. 2.11 Structure of Java 7

The JRE provides the JVM and other components necessary for you to *run* applets and applications written in the Java programming language (see Fig. 2.12).

Tools in JDK

The tools available in JDK are split into the following categories:

- Basic tools (javac, java, javadoc, apt, appletviewer, jar, jdb, javah, javap, extcheck)
- Security tools (keytool, jarsigner, policytool, kinit, klist, ktab)
- Internationalization tools (native2ascii)
- Remote Method Invocation (RMI) tools (rmic, rmiregistry, rmid, serialver)
- Java IDL and RMI-IIOP tools (tnameserv, idlj, orbd, servertool)

- Java deployment tools (pack200, unpack200)
- Java plug-in tools (html converter)
- Java Web Start tools (javaws)
- Java Monitoring and Management Console (jconsole)
- Java Web Services tools (schemagen, wsgen, wsimport, xjc)

Basic Tools

Figures 2.12 and 2.13 show the structure of Java and the basic tools available in JDK, respectively.

javac Java complier is named `javac`. The Java compiler takes input source code files (these files typically have the extension `.java`) and converts them into compiled bytecode files (these files have the extension `.class`).

java The Java interpreter, known eponymously as java, can be used to execute Java applications. The interpreter translates bytecodes directly into program actions.

javadoc As programmers, we have fought it in every way possible. Unfortunately, there is no longer any excuse for not documenting our source code. Using the `javadoc` utility provided with the JDK, you can easily generate documentation in the form of HTML files. To do this, you embed special comments and tags in your source code and then process your code through `javadoc`. All the online Java API documentation was created with `javadoc`.

apt It stands for Annotation Processing Tool, used for processing annotations.

```
Java API
  -User Interface Toolkits
  -Integration Libraries
  -Other Base Libraries
  -Lang and util Base Libraries

JRE
  -Deployment Technologies
  -Java API
  -Java Virtual Machine

JDK
  -Java Language Constructs
  -Tools and Tool APIs
  -JRE
```

Fig. 2.12 Java Structure

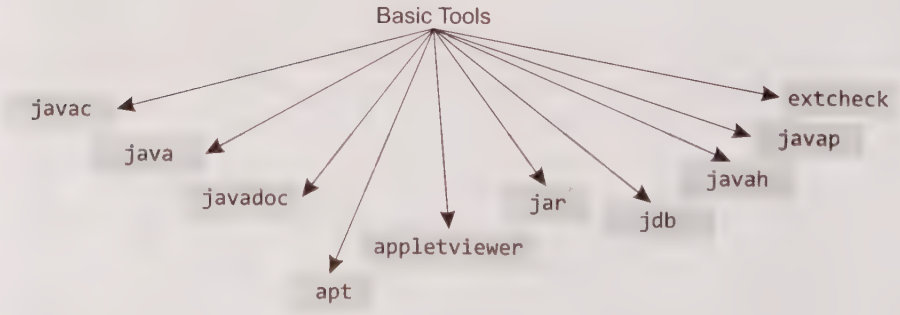


Fig. 2.13 Basic Tools Available in JDK

`appletviewer` This small program provides a real Java environment for testing applets. It loads the HTML file in which the applet has been embedded and displays the application in a browser-like window.

`jar` It is used for creating and managing jar (similar to WinZip file) files.

`jdb` The Java debugger, `jdb`, enables you to debug your Java classes. Unfortunately, the Java debugger is a throwback to the pre-GUI debugger dark ages of programming. The Java debugger is a command-line debugger. You can use the `jdb` to set breakpoints, inspect objects and variables, and monitor threads.

`javah` Because Java is a new language and must fit in a world dominated by C and C++, it includes the capability to use native C code within a Java class. One of the steps in doing this is by using the Java header file generator, `javah`.

`javap` One of the basic tenets of object-oriented programming is that programmers unfamiliar with a class need only concern themselves with the public interface of that class. If you want to use a class, you shouldn't be concerned with how this class has been written.

Because you should be interested only in the public interface of a class, the JDK includes a disassembler, `javap`, that can be used to display the public interface, both methods and variables, of a class. Additionally, the Java disassembler includes options to display private members or to display the actual bytecodes for the class's methods. This last option can be particularly useful if you want to achieve a greater understanding of the bytecodes used by the Java interpreter.

`extcheck` It is used for detecting Jar conflicts.

2.11 INTEGRATED DEVELOPMENT ENVIRONMENT

Integrated development environment (IDE) contains the tools specifically designed for writing Java codes. These tools offer a GUI environment to compile and debug your Java program easily from the editor environment as well as browse through your classes.

New Java IDEs are released every now and then, as Java is accepted as a viable programming language. Some of these IDEs are listed below.

Eclipse It is an open source extensible IDE. At present, it is a Java IDE and includes Java development tools. The requirement is that you should have the JRE installed on your machine. The IDE supports Windows XP, Windows 2000, Windows 7, Vista, Linux, and Solaris.

Gel It is an IDE for Java that features syntax highlighting (Java, JSP, HTML, XML, C, C++, Perl, Python, etc.), unlimited undo and redo, column selection mode, block indent and un-indent, highlighting of matching braces, spell-checking, automatic positioning of closing braces, auto indent, regular expression searches, find in files, code completion (Java and JSP), parameter hints, identifier hints, context-sensitive help linked to Javadoc, class browser, project management, integrated support for ANT and JUnit, differencing tool to compare files, etc. It works only on Windows.

DrJava It is an integrated development environment for Java, released under the GNU GPL that allows you to interactively evaluate Java expressions.

JCreator The light edition of this IDE for Java has support for project management, a syntax highlighting editor, wizards, class viewer, package viewer, tabbed documents, JDK profiles (which allows you to work with multiple JDK), a customizable user interface, etc. JCreator runs on Windows 95, 98, NT, and 2000.

NetBeans It is a cross-platform open source IDE for Java. It comes with a code editor that supports code completion, annotations, macros, auto-indentation, etc. It integrates with compilers, debuggers, JVMs, and other tools.

SUMMARY

Java is a programming language invented by James Gosling and others in 1994. Java was originally named Oak and was developed as a part of the Green Project at the Sun Company. Patrick Naughton, Mike Sheridan, and James Gosling were trying to figure out the next wave in computing and that wave came in 1995, when Java started to be visualized as a language for Internet applications.

It is conceived that Java is a pure object-oriented language, meaning that the outermost level of data structure in Java is the object. Java is designed to be platform independent, so it can run on multiple platforms. The same runtime code can be downloaded on any platform and be executed there, if that platform supports the Java runtime environment. For this, Java incorporates elements of both interpretation and compilation.

At the heart of Java Runtime Environment lies the Java Virtual Machine or JVM. Most programming languages compile source codes directly into machine codes, suitable for execution on a particular microprocessor architecture. But Java is somewhat different, as it uses bytecode—a special type of machine code. Java bytecode executes on a special type of microprocessor. As there was no hardware implementation of this microprocessor available when Java was first released, the complete processor architecture was emulated by a software known as the virtual machine.

Java is a robust language, as its two properties, type checking and interpretation makes Java programs crash-proof. Java has several other features that protect the integrity of the security system and prevent several common attacks. Java is inherently multithreaded, i.e., multiple threads developed in this language can be executed concurrently.

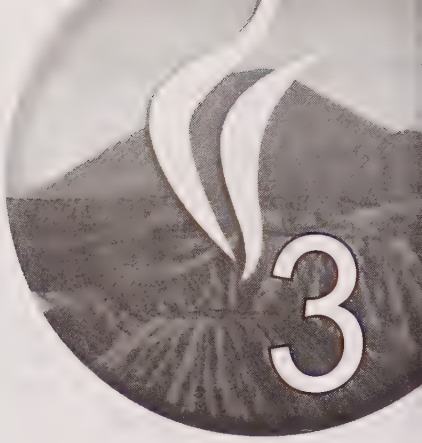
Other features of Java include automatic memory management, dynamic binding, optimal performance, built-in networking capabilities, etc. The garbage collector relieves the programmers from memory deallocation. Java uses references instead of pointers.

Every Java program consists of one or more classes. A class is nothing but a template for creating objects. In Java, codes reside inside a class. The name of the class must match with the name of the file.

EXERCISES

Objective Questions

1. What was the name of first version of Java?
   (a) Oak (b) Mustang
   (c) Tiger (d) Playground
2. What was the name of the team that developed Java?
   (a) Green Team (b) Star Seven
   (c) Sun (d) Java team
3. What is the name of the tool that is used for compiling a Java program?
   (a) javap (b) java
   (c) javah (d) javac
4. What is the name of the tool that is used for interpreting a Java program?
   (a) javap (b) java
   (c) javah (d) javac
5. What process automatically removes objects that are not being referenced?
   (a) Multithreading (b) Object Reclamation



Java Programming Constructs

I often say . . . that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be. **Lord Kelvin**

After reading this chapter, the readers will be able to

- understand how variables are used in
- know the basic data types
- learn expressions and conditional statements
- use all the available operations in Java
- know the basics of conversion and casting
- understand loops and branching statements

3.1 VARIABLES

Variable is a symbolic name refer to a memory location used to store values that can change during the execution of a program. Java declares its variables in the following manner:

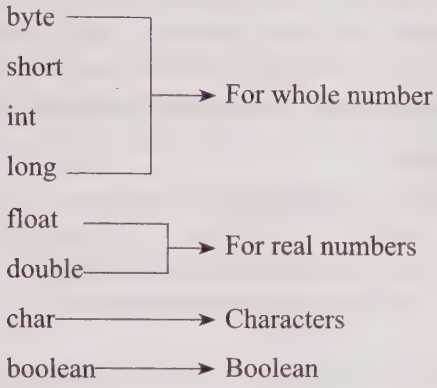
```
int       noofwatts   =    100;   // variable declaration
```

A variable declaration involves specifying the type (data type), name (identifier), and value (literal) according to the type of the variable. Let us have a look at the three components in detail.

3.2 PRIMITIVE DATA TYPES

Primitive data types are the basic building blocks of any programming language. A primitive data type can have only one value at a time and is the simplest built-in form of data within Java.

All variables in Java have to be declared before they can be used, that is why Java is termed as a *strongly typed language*. There are eight primitive data types in Java, as follows:



Java is portable across computer platforms. C and C++ leave the size of data types to the machine and the compiler, but Java specifies everything.

Note All integer (byte, short, int, long) and floating-point types (float, double) are signed in Java.

byte It is a 1-byte (8-bit) signed 2's complement integer. It ranges from –128 to 127 (inclusive). The **byte** data type can be used where the memory savings actually matter.

short It is a 2-byte (16-bit) signed 2's complement integer. It ranges from –32,768 to 32,767 (inclusive). As with **byte**, you can use a **short** to save memory.

int It is a 4-byte (32-bit) signed 2's complement integer. It ranges from –2,147,483,648 to 2,147,483,647 (inclusive). For integral values, this data type is the default choice.

long It is an 8-byte (64-bit) signed 2's complement integer. It ranges from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (inclusive). This data type should be used only when you need a range of values wider than **int**.
Floating point conforms to the IEEE 754-1985 binary floating point standard.

float It is a single-precision 32-bit floating point. It ranges from 1.401298464324817e–45f to 3.402823476638528860e+38f.

double This data type is a double-precision 64-bit floating point. It ranges from 4.94065645841246544e–324 to 1.79769313486231570e (+) 308. For decimal numbers, this data type is the default choice.

boolean It has only two possible values: **true** and **false**. The size of this data type is not precisely defined.

char The unsigned **char** data type is a single 16-bit unicode character. It ranges from '\ **u0000'** (or 0) to **'\uffff'** (or 65,535 inclusive).

Note Unlike C/C++, where handling of character sequences is tedious, Java provides a class named "String" for handling character strings enclosed within double quotes. Although it is not a primitive data type, Java string solves much of the complexity with ease.

3.3 IDENTIFIER

Identifiers are names assigned to variables, constants, methods, classes, packages, and interfaces. No limit has been specified for the length of a variable name. Identifiers can have letters, numbers, underscores, and any currency symbol. However they may only begin with a letter, underscore, or a dollar sign. Digits cannot be the first character in an identifier.

3.3.1 Rules for Naming

1. The first character of an identifier must be a *letter*, an *underscore*, or a *dollar sign* ($).
2. The subsequent characters can be a *letter*, an *underscore*, *dollar sign*, or a *digit*. Note that *white spaces* are not allowed within identifiers.
3. Identifiers are *case-sensitive*. This means that *Total_Price* and *total_price* are different identifiers.
   Do not use Java's *reserved keywords*. A few examples of legal and illegal identifiers are shown below.

Legal Identifiers	Illegal Identifiers
`MyClass`	`My Class`
`$amount`	`23amount`
`_totalPay`	`-totalpay`
`total_Commission`	`total@commission`

3.3.2 Naming Convention

Names should be kept according to their usage, as it is meaningful and easy to remember as shown in Fig. 3.1.

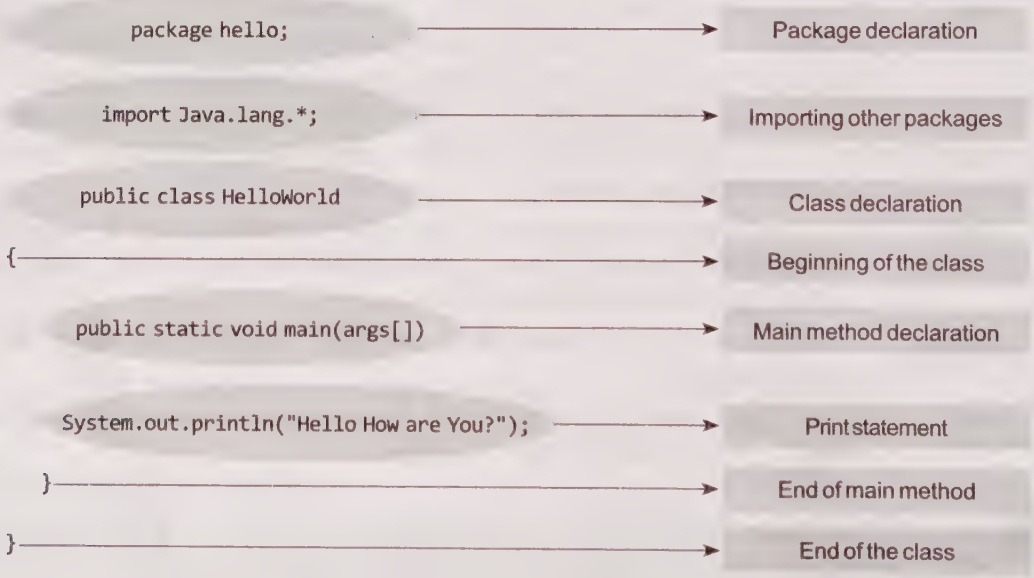


Fig. 3.1 Naming Convention Used in Java

Class or Interface Identifiers These begin with a capital letter. The first alphabet of every internal word is capitalized. All other letters are in lower case.

```
public class MyClass          // class identifier: MyClass
interface Calculator;         // interface identifier: Calculator
```

Variable or Method Identifiers These start with a lower-case letter. The first alphabet of every internal word is capitalized. All other letters are in lower case.

```
int totalPay;                                  // variable identifier: totalPay
MyClass.showResult();
// MyClass is the Class Name and showResult() is a method of MyClass.
```

Constant Identifiers These are specified in upper case. Underscores are used to separate internal words.

```
final double TAX_RATE = 0.05;                  // constant identifier: TAX_RATE
```

Package Identifiers These consist of all lower-case letters.

```
package mypackage.subpackage.subpackage;     //Package Declaration
```

3.3.3 Keywords

Keywords are predefined identifiers meant for a specific purpose and cannot be used for identifying used defined classes, variables, methods, packages, and interfaces. All keywords are in lower case. Table 3.1 lists the keywords in Java.

Table 3.1 Keywords in Java

abstract	assert	boolean	break	byte
case	catch	char	class	continue
default	do	double	else	enum
extends	final	finally	float	for
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while	const*	goto*
*const and goto are reserved keywords.				

3.4 LITERALS

A literal is a value that can be passed to a variable or constant in a program. Literals can be numeric (for byte, short, int, long, float, double), boolean, character, string notations or null literals.

Numeric Literals can be represented in binary, decimal, octal, or hexadecimal notations. These literals can be assigned to all numeric types in Java including char (based on their respective range and size).

Binary literals are a combination of 0's and 1's. Binary literals can be assigned to variables in Java 7. Binary literals must be prefixed with 0b or 0B (zerob or zeroB). For example,

```
char bin1 = 0b1010000;          // value in bin1 will be P
char bin2 = 0b1010001;          // value in bin2 will be Q
float bin3 = 0b1010000;         // value in bin3 will be 80.0
int bin4 = 0b1010001;           // value in bin4 will be 81
```

In case ***octal literals*** have to be specified, the value must be prefixed with a zero and only digits from 0 to 7 are allowed.

For example,

```
int x = 011;                    //value in x is 9
char y=0150;                    // value in y will be h
float z=0234;                   // value in z will be 156.0
```

Hexadecimal literals are prefixed with *0x* or *0X*; the digits 0 through 9 and *a* through *f* (or *A* through *F*) are only allowed. For example,

```
int y = 0x0001;                 //value in y is 1
char x=0x45;                    // value in x will be E
float y=0xA3;                   // value in y will be 163.0
```

All ***integer literals*** are of type **int**, by default. To define them as long, we can place a suffix of *L* or *l* after the number for instance:

```
long l = 2345678998L;
```

All ***floating literals*** are of type **double**, by default. To define them as float literals, we need to attach the suffix *F* or *f*. For double literals, *D* or *d* are suffixed at the end; however, it is optional. For instance,

```
float f = 23.6F;
double d = 23.6;
```

Java 7 onwards the readability of literals can be enhanced by using underscore with numeric literals. As the number of zeroes increase in a literal, counting the number of zeroes becomes tedious. In such big literals, underscores can be used as shown below:

```
int numlit=100_000_000;         // value in numlit will be 100000000
```

Underscores can be used not only with decimal literals but also with hexa, binary, and octal literals as shown below:

```
int numlit=0x100_000;               // value in numlit1 will be 1048576
int bin=0B1_000_000_000_000_001;    // vale in bin will be 32769
float octlit=03_000;                // value in octlit will be 1536.0
```

Note Underscore can only be used with literal values.

The following examples show some valid and invalid use of underscores.

```
int i =_23;                 // illegal, cannot start a literal with underscore
long f = 3_2_222_2_1;       // invalid use of underscore between value and suffix
long f = 3_2_222_21;        // legal
float e = 4_.2_3f;          // illegal use of underscore with a dot
```

```
float d = 4_2.2_3f;        // legal
float e = 4_2.2_3_f        // illegal
int i = 0_x_A_E;           // illegal use of underscore in prefix
int j = 0x_A_E;            // illegal use of prefix between prefix and literal
int k = 0xA_E;             // legal
```

For ***char literals***, a single character is enclosed in single quotes. You can also use the prefix **\u** followed by four hexadecimal digits representing the 16-bit unicode character:

```
char c = '\u004E'; char sample = 'A'; char example = 'a';
```

A single quote, a backslash or a unprintable character (such as a horizontal tab) can be specified as a character literal with the help of an escape sequence. An *escape sequence* represents a character by using a special syntax that begins with a single backslash character. Unicode is a type of escape sequence (refer Table 3.2). Furthermore, the syntax of unicode escape sequence consists of **\uxxxx** (where each **x** represents a hexadecimal digit).

Table 3.2 Unicode Escape Sequences to Represent Printable and Unprintable Characters

'\u0041'	Capital letter A
'\u0030'	Digit 0
'\u0022'	Double quote "
'\u003b'	Punctuation ;
'\u0020'	Space
'\u0009'	Horizontal Tab

Table 3.3 Special Escape Sequences

\\	Backslash
\"	Double quote
\'	Single quote
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab

For instance, `char c = '\t'; //` creates a character that represents horizontal tab (Refer Table 3.3).

Unicode characters can be assigned to strings in Java 7. Unicode 6, which has thousands of characters, cannot be accommodated in a 16-bit char data type. Increasing the size of char data type would lead to backward compatibility problems. To maintain compatibility with the application and standards, the string ("U+hex") is used to express unicode characters in Java.

A ***boolean literal*** is specified as either **true** or **false**. By default, it takes the value false (Refer Table 3.4). The following code fragment demonstrates a boolean literal:

```
boolean firstRoll = true;
```

String literals consist of zero or more characters within double quotes. For instance,

```
String s = "This is a String Literal";
```

Null literals are assigned to object reference variables (see Chapter 4 for object references).

```
s = null;
```

Table 3.4 shows a summary of the data types along with their respective default values, size, and range.

Table 3.4 Data Types: Size, Default Value, and Range

Data Type	Default Value	Size	Range
byte	0	8	–128 to 127 (inclusive)
short	0	16	–32,768 to 32,767 (inclusive)
int	0	32	–2,147,483,648 to 2,147,483,647 (inclusive)
long	0L	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (inclusive)
float	0.0F	32	1.401298464324817e–45f to 3.402823476638528860e+38f
double	0.0D	64	4.94065645841246544e–324 to 1.79769313486231570e+308
char	'\u0000'	16	0 to 65535
boolean	false	Not defined	true or false

Table 3.5 lists the reserved literals in Java.

Table 3.5 Reserved Literals.

true	false	null

3.5 OPERATORS

An operator performs an action on one or more operands. An operator that performs an action on one operand is called a *unary operator* (+, –, ++, – –). An operator that performs an action on two operands is called a *binary operator* (+, –, / , * , and more). An operator that performs an action on three operands is called a *ternary operator* (? :). Java provides all the three operators. Let us begin the discussion with binary operators.

3.5.1 Binary Operators

Java provides *arithmetic, assignment, relational, shift, conditional, bitwise,* and *member access* operators.

Assignment Operators

It sets the value of a variable (or expression) to some new value. The simple '=' operator sets the left-hand operand to the value of the right-hand operand. The assignment operator has right to left associativity (discussed in Section 3.7); so the statement `a = b = 0;` would assign 0 to `b` then `b` to `a`. Java supports the following list of *shortcut* or *compound* assignment operators:

```
+= -= *= /= %= &= |= ^= <<=>>= >>>=
```

These operators allow you to combine two operations into one: one fixed as assignment plus another one. These operators will be explained shortly according to their counterparts.

Arithmetic Operators

Arithmetic operators are used for adding (+), subtracting (–), multiplying (*), dividing (/), and finding the remainder (%).

Java does not support operator overloading. There are certain languages like C++ that allow programmers to change the meaning of operators enabling them to act in more than one way depending upon the operands. But there are certain operators which are overloaded by Java itself

like + operator which behaves differently when applied to different operands. For example, if + operator is used and one of the operands is a string, then the other operand is converted to a `String` automatically and concatenated. It is evident in the following examples in the `System.out.println` statement when the `String` in the quotes is concatenated with the values of the result or individual primitives. In addition to these operators, arithmetic compound assignment operators are also provided by Java: +=, – =, /=, *=, %=. For example,

```
a += b;              // evaluated as a = a + b;
a -= b;              // evaluated as a = a - b;
a *= b;              // evaluated as a = a * b;
a /= b;              // evaluated as a = a / b;
a % = b;             // evaluated as a = a % b
```

Let us take an example to demonstrate the use of these operators in Java. A close look at the program will show us that the + operator can be used for two purposes: concatenation and addition in the print statements.

Example 3.1 Demonstration of Arithmetic Operators

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

     int a = 25, b = 10;
     System.out.println("Sum "+ a +" +" + b +" = " + (a + b));
     //adding two variables a and b

     System.out.println("Subtraction "+ a +" - " + b +" = " + (a - b));

     //multiplying a with b
     System.out.println("Multiplication "+ a +" * " + b +" = " + (a * b));

     // Division
     System.out.println("Division "+ a +" / " + b +" = " + (a / b));

     // Remainder Operator
     System.out.println("Remainder "+ a +" % " + b +" = " + (a % b));

     // a and b can be added and the result can be placed in a
     // Let us see how?
     a += b;
     System.out.println("Added b to a and stored the result in a" + a);
  }
}
```

Output

```
C:\Javaprg\>Java ArithmeticDemo
Sum 25 + 10 = 35
Subtraction 25 - 10 = 15
Multiplication 25 * 10 = 250
Division 25 / 10 = 2
Remainder 25 % 10 = 5
Added b to a and stored the result in a 35
```

Figure 3.2 shows how the '+' operator concatenates and adds the operands

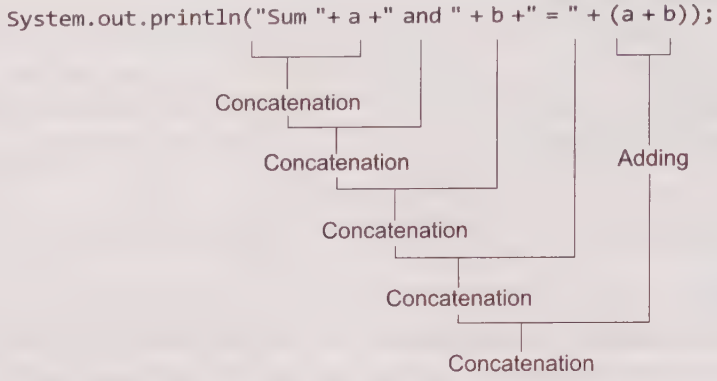


Fig. 3.2 The Operation of '+' Operator

Relational Operators

Relational operators in Java return either *true* or *false* as a boolean type. Table 3.6 shows a list of all the relational operators in Java.

Relational operators in C++ returns an integer where the integer value of zero may be interpreted as false and any non-zero value may be interpreted as true.

Table 3.6 Relational Operators

equal to	==
Not equal to	!=
less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=

Example 3.2 Demonstration of Relational Operators

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

     int a = 10,b = 20;
     System.out.println("Equality Operator: a == b : \t\t\t" +(a == b));
     System.out.println("Not Equal To Operator: a != b : \t\t" +(a != b));
     System.out.println("Less Than Operator: a == b : \t\t\t" +(a < b));
     System.out.println("Greater Than Operator: a == b : \t\t" +(a > b));
     System.out.println("Less than or equal to Operator: a == b : \t" +(a <= b));
     System.out.println("Greater than or equal to Operator: a == b : \t" +(a >= b));
     }
}
```

Output

```
C:\Javaprg\>Java RelationalOperatorDemo
Equality Operator: a == b :                  false
Not Equal To Operator: a != b :              true
Less Than Operator: a == b :                 true
Greater Than Operator: a == b :              false
Less than or equal to Operator: a == b :     true
Greater than or equal to Operator: a == b : false
```

Boolean Logical Operators

Boolean logical operators are: conditional OR (||), conditional AND (&&), logical OR (|), logical AND (&), logical XOR (^), unary logical NOT (!). Boolean logical operators are applied to boolean operands or expressions (Section 3.6) and return a boolean value. The bitwise logical AND (&), logical OR (|), logical XOR (^) and logical NOT (~) operators are applied to integers to perform bitwise logical operations discussed later.

Logical OR results in true if one of the operands is true. **Logical AND** results in false if one of the operands is false. **Logical XOR** works like OR with an exception, that is, in case if both the operands of an XOR operator are true then the answer is false. **Logical NOT** is just the compliment of the boolean operand.

Conditional OR (||) and AND (&&) operators also known as *short-circuit operators* conditionally evaluate the second operand or expression. In case of OR, if the first operand is true, no matter what the second operand is, the answer is true. In case of AND, if the first operand is false, no matter what the second operand is, the answer is false. So there is no need to evaluate the second operand.

In addition to these operators, boolean compound assignment operators are also provided by Java: &=, |=, ^=. For example,

```
a &= b;      // evaluated as a = a & b;
a |= b;      // evaluated as a = a | b;
a ^= b;      // evaluated as a = a ^ b;
```

Example 3.3 Demonstration of Boolean Operators

```
class BooleanLogicalOperatorDemo
{
    public static void main(String args[])
    {
    boolean a = true,b = false;
    System.out.println("Logical OR: "+ a +" | "+b+": " +(a|b));
    System.out.println("Logical XOR: "+ a +" ^ "+b+": "+(a^b));
    System.out.println("Logical AND: "+ a +" & "+b+": "+(a&b));
    System.out.println("Logical NOT: !a : "+(!a));
    System.out.println("Conditional OR: "+ a +" || "+b+": "+(a||b));
    System.out.println("Conditional AND: "+ a +" && "+b+": "+(a&&b));
    // shortcut operator
    a |= b;
    System.out.println("Shortcut OR: "+ a +" | "+b+" = "+(a));
    }
}
```

Output

```
Logical OR: true | false: true
Logical XOR: true ^ false: true
Logical AND: true & false: false
Logical NOT: !a : false
Conditional OR: true || false: true
Conditional AND: true && false: false
Shortcut OR: true | false = true
```

Bitwise Operators

Bitwise operators include *and, or, xor, not, right shift, left shift,* and *unsigned right shift*. In Java, *bitwise* operators operate on `int` and `long` values. If any of the operand is shorter than an `int`, it is automatically promoted to `int` before the operations are performed (see Section 3.8). Table 3.7 lists the bitwise operators and how they function.

Note It is important to understand how integers are represented in binary. For example, the decimal number 4 is represented as 100 in binary and 5 is represented as 101. Negative integers are always represented in 2's complement form. For example, –4 is 1111 1111 1111 1111 1111 1111 1111 1100.

Bitwise shortcut operators are used in the same way as boolean operators. Bitwise shortcut operators include the following:

```
AND: &=
OR: |=
XOR: ^=
Shift Operator: >>=,<<=,>>>=
```

Table 3.7 Bitwise Operators

`a & b`	1 if both bits are 1
`a | b`	1 if either of the bits is 1
`a ^ b`	1 if both bits are different
`~a`	Complement the bits.
`a << b`	Shift the bits left by b positions. Zero bits are added from the LSB side. Bits are discarded from the MSB side.
`a >> b`	Shift the bits right by b positions. Sign bits are copied from the MSB side. Bits discarded from the LSB side.
`a >>> b`	Shift the bits right by b positions. Zero bits are added from the MSB side. Bits are discarded from the LSB side.

Example 3.4 Demonstration of Bitwise Operators

```
class BitwiseOperatorDemo
{
     public static void main(String args[])
     {
       int x = 2,y = 3;
       System.out.println("Bitwise AND: " +x+ "&" +y+ " = " +(x&y));
```

```
        System.out.println("Bitwise OR :" +x+ "|" +y+ " = " +(x|y));
        System.out.println("Bitwise XOR:" +x+ "^" +y+ " = " +(x^y));
        System.out.println("Bitwise NOT: ~" +x+ "=" +(~x));
    }
}
```

Output

```
Bitwise AND: 2&3 = 2
Bitwise OR : 2|3 = 3
Bitwise XOR: 2^3 = 1
Bitwise NOT: ~2=-3
```

Shift operators shift the bits depending upon the type of operator. The left shift operator shifts the numbers of bits specified towards the left. Bits are discarded from the left and added from the right with the value of bits being zero. The right shift operator shifts the numbers of bits specified towards right. Bits are discarded from the right and added from the left side with the value of bits being that of the sign bit. The unsigned right shift shifts the numbers of bits specified towards right. Bits are discarded from the right and added from the left side with the value of bits being zero. For example, let us assume $x = 4$ and this x is to be shifted by the shift distance of 1.

```
int y = x >> 1;
//y has the value 2, value is halved in each successive right shift
      = 00000000 00000000 00000000 00000100 >> 1
      = 00000000 00000000 00000000 00000010 (which is 2)
int y = x << 1;
// y has the value 8, value is doubled in each successive left shift

int y = x >> 1;       // same as right shift for positive numbers.
```

If we provide a negative number to be left or right shifted, then the negative numbers are represented in 2's compliment arithmetic and then shifted. If we provide an `int` negative shift distance as shown in the following example, first the negative shift distance is ANDed with the mask 11111 (i.e., 31) and the result is the new shift distance. If we provide a long negative shift distance as shown in the following example, first the negative shift distance is ANDed with the mask 111111 (i.e., 63), and the result is the new shift distance.

Example 3.5 Shift Operators

```
class ShiftOperatorDemo
{
    public static void main(String args[])
    {
      int x = 5,y = 1;
      System.out.println("Left shift: "+x+"<<"+y+"="+(x<<y));
      System.out.println("Right shift: "+x+" >> "+y+"="+(x >> y));
      System.out.println("Unsigned Right Shift: "+x+" >>> "+y+"="+(x >>> y));

      //negative numbers
```

```
        System.out.println("Right Shift: -"+x+" >> "+y+"="+(-x >> y));
        System.out.println("Unsigned Right Shift: -"+x+" >>> "+y+"="+(-x >>> y));
        System.out.println("Left shift: -"+x+" << "+y+"="+(-x << y));

        //negative shift distance of -31 actually means shifting 1 bit
      System.out.println("Left shift:"+x+"<<-31 ="+(x << -31));
      }
}
```

Output

```
Left shift: 5 << 1 = 10
Right shift: 5 >> 1 = 2
Unsigned Right Shift: 5 >>> 1 = 2
Right Shift: -5 >> 1 = -3
Unsigned Right Shift: -5 >>> 1 = 2147483645
Left shift: -5 << 1 = -10
Left shift: 5 << -31 = 10
```

Bitwise operators are particularly used where bit-level or low-level programming is required such as writing device drivers, working with embedded systems, compression of data, encryption and decryption of data, setting mask and flags, and creating networking protocols for communication.

3.5.2 Unary Operators

Unary operators, as the name suggest, are applied to only one operand. They are as follows: ++, - -, !, and ~. The unary boolean logical `not` (!) and bitwise logical `not` (~) have already been discussed.

Increment and Decrement Operators

Increment and decrement operators can be applied to all integers and floating-point types. They can be used either in prefix (– –x, ++x) or postfix (x– –, x++) mode.

Prefix Increment/Decrement Operation

```
int x = 2;
int y = ++x; // x = 3, y = 3
int z = --x; // x = 1, z = 1
```

Postfix Increment/Decrement Operation

```
int x = 2;
int y = x++; // x == 3, y == 2
int z = x--; // x = 1, z = 2
```

We will discuss these operators in Example 3.6.

3.5.3 Ternary Operators

Ternary operators are applied to three operands. This conditional operator (? :) decides, on the basis of the first expression, which of the two expressions to be evaluated.

```
operand1 ? operand2 : operand3
```

operand1 must be of boolean type or an expression producing a boolean result. If operand1 is true, then operand2 is returned. If operand1 is false, then operand3 is returned. This operator is similar to an if conditional statement. For example,

```
String greater = x < y ? "Y is greater" : " X is greater";
```

If the value of x is less than y, "Y is greater" string is retuned and stored in the variable: greater, else "X is greater" is retuned and stored in the variable: greater.

3.6 EXPRESSIONS

An *expression* is a combination of operators and/or operands. Java expressions are used to create objects, arrays, pass values to methods and call them, assigning values to variables, and so on. Expressions may contain identifiers, types, literals, variables, separators, and operators (we have already discussed all these topics). For example,

```
int m = 2,n = 3,o = 4;
int y = m * n * o;
```

m=2 is an expression which assigns the value 2 to variable m. Similarly, n=3 and o=4 are expressions where n and o are being assigned values 3 and 4. m * n * o is also an expression wherein the values of m, n, and o are multiplied and the result is stored in the variable y.

3.7 PRECEDENCE RULES AND ASSOCIATIVITY

Precedence rules are used to determine the order of evaluation priority in case there are two operators with different precedence. Associativity rules are used to determine the order of evaluation if the precedence of operators is same. Associativity is of two types: Left and Right. *Left associativity* means operators are evaluated from left to right and vice versa for right associativity. Precedence and associativity can be overridden with the help of parentheses.

Table 3.8 Precedence Rule and Associativity

Operators	Associativity
., [], (args), i++, i-¬	L R
++i, --i, +i, -i, ~, !	R L
new, (type)	R L
*, /, %	L R
+, ¬	L R
<<, >>, >>>	L R
<, >, <=, >=, instanceof	Non Associative
= =, ! =	L R
&	L R
^	L R
|	L R
&&	L R
| |	L R
? :	R L
=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, |=	R L

Table 3.8 lists the operators in Java according to their precedence (from highest to lowest) and their respective associativity's. Operators in a row have same precedence.

Here L → R indicates associativity from left to right and R → L indicates associativity from right to left.

Example 3.6 Precedence Rules

```
class AssociativityAndPrecedenceTest
{
public static void main(String[] args)
{
//precedence of * is more than that of +

L1      System.out.println(" 2 + 3 * 2 = \t " + (2 + 3 * 2));

        //Associativity applies in case of operators with equal
        //Precedence. below is a case of Left Associativity
L2      System.out.println(" 2 * 5 / 3 = \t " + (2 * 5 / 3 ));

        // Precedence overridden with help of parentheses
L3      System.out.println("(2 + 3) * 2 = \t " + ((2 + 3) * 2));
        int x;
        int y = 3;
        int z = 1;
        //Assignment associates from right to left

L4      x = y = z;
L5      System.out.println(" x = y = z: \t" + x);

        //+ and - have left associativity
L6      System.out.println(" 3 - 2  + 1 = \t " + (3 - 2 + 1));

        //evaluating long expressions to check Precedence and Associativity
        int i = 10;
        int j = 0;
        int result = 0;

L7      result = i-- + i / 2 - ++i + j++ + ++j;
        System.out.println("i: " +i+ " j " +j+ " result: "+result );

        // + operator has a left to right associativity
L8      System.out.println("Hello "+1+2);

        // First two numbers are added and the added result is concatenated with
        // String "Hello"

L9      System.out.println(1+2+" Hello");

        }
}
```

Output

```
C:\Javabook\programs\chap3>Java AssociativityAndPrecedenceTest
2 + 3 * 2 = 8
```

```
2 * 5 / 3 = 3
(2 + 3) * 2 = 10
x = y = z: 1
3 - 2 + 1 = 2
i: 10 j 2 result: 6
Hello 12
3 Hello
```

Explanation

L1 Shows the precedence of * is more than +, that is why 3 is first multiplied with 2 and the result (6) is added with 2 and then printed.

L2 Shows two operators with equal precedence, * and /. In this case, associativity plays a role instead of precedence. As is evident from Table 3.9, * and / have left associativity, so the operators will be evaluated from the left side. That is why 2 is multiplied with 5 first and then the result (10) is divided by 3 to give the integer quotient 3, which is then printed.

L3 Shows the precedence of (nudge) is more than * and + (or any other operator, refer Table 3.9). In this case, operation within parentheses is performed first, that is, 2 is added to 3 and then the result (5) is multiplied with 2 to give 10 which is then printed. Also note that when no parentheses were used in L1, the answer was 8.

L4 Shows the assignment operator which is right associative, so first the value of `z` is assigned to `y` and then the value of `y` is assigned to `x`.

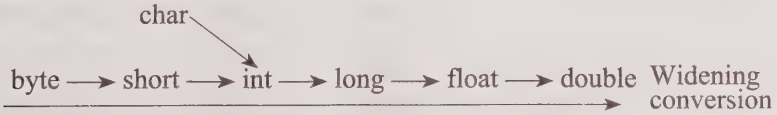
L6 Portrays the case of same precedence, so associativity is used for expression evaluation. Table 3.9 shows + and – have left associativity, so 2 is subtracted from 3 first and then the result (1) is added to 1 to output 2, which is then printed on the screen.

L7 In this expression, `i-- + i / 2 - ++i + j++ + ++j`, the decremented value of `i` will be reflected while evaluating the sub-expression `i/2`, i.e., `i--+i/2` will be evaluated as 10 + 9/2 (result of sub-expression is 14). At this point `i` will have the value 9. While evaluating `++i`, the value of `i` is incremented first and then added, so the value of i becomes 10 again (result of expression at this point is 14 –10 = 4). The value of `j` (i.e. 0) is added to the expression first and then incremented in the sub-expression `j++`. Now `j` has the value 1 (result of expression at this point `i-- + i/2 - ++i + j++`is 4). In the last sub-expression `++j`, the value of `j` (which is 1 now) is incremented first and then added to the expression (value of `j` is now 2 which is added to 4 to produce 6 as the result).

L8 & 9 Show the usage of + operator between different operands. The important point to note is that associativity and not precedence will be used for evaluating expression. The associativity of + operator is from left to right, so the String "Hello" is concatenated to 1 first and then String "Hello 1" is concatenated to the second number 2. In L9, the numbers are added first and then the sum is concatenated with the String.

3.8 PRIMITIVE TYPE CONVERSION AND CASTING

In Java, type conversions are performed automatically when the type of the expression on the right-hand-side of an assignment operation can be safely promoted to the type of the variable on the left-hand-side of the assignment.



Conversions that are implicit in nature are termed as *widening conversions*. In an assignment statement, the types of the left-hand-side and right-hand-side must be compatible. If the right-

`sign bit * mantissa * 2^exponent`

Java uses a radix of 2. A float variable has 23 bits for mantissa and 8 bits for exponent. A `double` variable uses 52 bits for mantissa and 11 bits for exponent. The bit representation of these variables is shown below:

Sign bit	8 exponent bits	23 mantissa bits

float variable

Sign bit	11 exponent bits	52 mantissa bits

double variable

So you can easily imagine that a `float` variable can accommodate a lot more values that what a long variable can because of its representation and format. For a more detailed discussion on floating point standard refer to IEEE 754 floating point standard.)

Let us take an example to understand the concepts.

Example 3.7 Conversion and Casting

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

        //casting
L1      int i = (int)(8.0/3.0);
        // j will have the largest value of its type as 2147483648.0f is too large
L2      int j = (int)2147483648.0f;
        System.out.println("i = " +i+ " j =" +j);

        //casting: answer will contain 8 low order bits of the int value of 257
L3      byte b = (byte)257;
        //casting: answer will contain 16 low order bits of the int value of 65537
L4      short s = (short)65537;
        System.out.println("b =" +b+ " s ="+s);

        //casting int to char
L5      System.out.println("Converting int to char " +(char)75);

        //conversion: int * byte * short * double is double
L6      double d = i * b * s * 2.0;
        System.out.println("Conversion to double result is : "+d);

        //implicit conversion to int in case of shift operator
L7      i = b << 2;
        System.out.println("i = "+i);
        // compound operator automatically perform casting
        byte c = 0;
```

```
L8          //c = c + b;    does not compile
L9          c += b;         // complies
            System.out.println("Result: "+c);
        }
    }
```

Output

```
i = 2 j = 2147483647
b = 1 s = 1
Converting int to char K
Conversion to double result is : 4.0
i = 4
Result: 1
```

Explanation

L1 It shows the casting of a `double` expression into an `int`. The result of dividing a `double` value by a `double` value is a `double`, which is then casted into an `int`.

L2 It shows the casting of a `float` literal into an `int`, which is larger than the maximum value an `int` variable can hold. So `j` is set to the maximum value an `int` can hold.

L3 It shows the casting of an `int` literal into a `byte`. It is again larger than the maximum a `byte` can hold. In this case, `byte` variable will contain the value which is present in the 8 low order bit of the `int` literal 257. The `int` literal 257 has the binary value 00000000 00000000 00000001 00000001. After casting, `byte` will have the low order 8 bits (00000001), which is the decimal value 1.

L4 It shows the casting of an `int` literal into a `short`, which is larger than the maximum a short can hold. In this case, the `short` variable will contain the value that is present in the 16 low order bits of the `int` literal 65537. `int` literal 65537 has the binary value 00000000 00000001 0000000 00000001. After casting, `short` will have the low order 16 bits (00000000 00000001) which is the decimal value 1.

L5 It shows the casting of an `integer` into a `char`. Characters are represented by integral ASCII values, which can be casted back to character. An integer variable can hold a character, e.g. `int x ='K'`. This is a case of automatic promotion; here `x` will have the value 75 which is the ASCII value of `'K'`.

L6 It shows the multiplication automatic promotion. The expression involves multiplication that is left associative. First, `byte` variable `b` is automatically promoted to `int` and multiplied with `i` giving an `int` result (i.e., 2*1 = 2). Then `short` is automatically promoted to an `int` and multiplied with the previous `int` result to give a new `int` result (i.e., 2*1 = 2). Now this `int` result is automatically promoted to `double` because it has to be multiplied to a `double` literal (i.e., 2.0) giving a `double` result of 4.0.

L7 It shows the left shifting of bits in the expression `b << 2`. Before shifting, there is an automatic promotion of `byte` variable `b` to an `int` and then the 32 bits are shifted towards left by two places.

L8 It is commented, as it will not compile. During evaluation of this expression `c` and `b` are automatically promoted to `int` and added to produce an `int` result. This result cannot be stored directly in a byte variable.

L9 It complies because the operator used in this case is a compound operator that automatically casts the result into the destination type.

3.9 FLOW OF CONTROL

Control flow statements help programmers make decisions about which statements to execute and to change the flow of execution in a program. The four categories of control flow statements available in Java are *conditional statement, loops, exception,* and *branch.*

3.9.1 Conditional Statements

Java programs accomplish their tasks by manipulating the program data using operators and making decisions by testing the state of program data. When a program makes a decision, it determines, based on the state of the program data whether certain lines of code should be executed. For example, a program may examine a variable called *flag* to determine if it should execute a block of code that saves data into a file on to the disk. If flag is true, the data is saved; else the data is not saved. The two conditional statements provided by Java are: `if … else` and `switch-case`.

if…else

The syntax of `if` statement is as follows:

```
if (x = = 0)
     {// Lines of code}
else if(x = = 1)
     {// Lines of code}
.....
else
     {// Lines of code}
```

The arguments to a conditional statement like `if` must be a boolean value, which is something that evaluates to true or false. You can have *n* number of `else if (){}` statements in your program, as per your requirement. The `if...else` condition can also be nested as shown.

```
if (condition)
{
     if (condition)
          {//do something based on the condition}
}
```

The following example shows how `if...else` conditional statements can be used in Java.

Example 3.8 **if...else**

```
class IFElseExample
{
    public static void main(String args[])
    {
        int x=20,y=18,z=22;
L1      if (x < y)                    // x comes before y
L2      {
L3      if (z < x)                    // z comes first
L4      System.out.println( z + " " + x + " " + y);
L5      else if (z > y)               // z comes last
L6      System.out.println(x + " " + y + " " + z);
L7      else                          // z is in the middle
L8        System.out.println(x + " " + z + " " + y);
L9      }
L10     else
```

```
L11          {                              // y comes before x
L12          if (z < y)                     // z comes first
L13          System.out.println(z + " " + y + " " + x);
L14          else if ( z > x)               // z comes last
L15          System.out.println(y + " " + x + " " + z);
L16          else                           // z is in the middle
L17          System.out.println(y + " " + z + " " + x);
           }
       }
   }
```

Output

```
C:\>Java IFElseExample
18 20 22
```

Explanation

L1 It shows `if` statement comparing `x` with `y`. If `x` is less than `y`, then control passes into the enclosing curly brackets starting from L2. But in our example, `x` is greater than `y`, so the control passes to the `else` statement in L10.

L3 It uses the nested `if` statement. This `if` clause is within the `if` statement on L1. If condition in L1 returns true, then the condition on this line is checked. The condition checks whether `z` is less than `x`, which is already less than `y` from L1. If `(z < x)` is true, then `z` is the smallest of the three, `y` is the largest, and `x` lies in between.

L4 It prints the facts of L3.

L5 If condition on L3 returns false, the control passes on to L5, which means `z` is not less than `x` and in L5, `z` is compared with `y`. If `z` is greater than `y`, it means `z` is the largest, `x` is smallest, and `y` lies in between. (We already know the fact from L1 that `x` is less than `y`).

L6 It prints the facts of L5.

L7 If condition on L5 returns false, then the control passes on to the `else` on L7, which means that `x` is less than `y` (L1) and `z` is not less than `x` (L3) and is not greater than `y` (L5). So `x` is the smallest, `y` is the largest, and `z` lies in between.

L8 It prints the facts of L7.

L10 The `else` of `if` on L1. The control passes on to this `else` if L1 returns false, which means `x` is not less than `y`.

L11 The starting curly bracket of `else`.

L12 It checks if `z` is less than `y`. If true, `z` comes first, then `y`, and `x` is the largest.

L13 It prints the facts of L12.

L14 It checks if `z` is greater than `x`. If true, `y` comes first, then `x`, and `z` is the largest. In our example, this case is executed as the value of `x` is 20, `y` is 18, and `z` is 22.

L15 It prints the facts of L14.

L16 If `z` is not less than `y` (L12) and `z` is not greater than `x`, i.e., `z` is in the middle, `y` is the smallest, and `x` is the largest.

L17 It prints the facts of L16.

Switch-case

Java has a shorthand for multiple `if` statement—the `switch-case` statement. Here is how we can write the above program using a `switch-case`:

```
switch (x) {

case 0:
     // Lines of code
```

3.9.1 Conditional Statements

Java programs accomplish their tasks by manipulating the program data using operators and making decisions by testing the state of program data. When a program makes a decision, it determines, based on the state of the program data whether certain lines of code should be executed. For example, a program may examine a variable called *flag* to determine if it should execute a block of code that saves data into a file on to the disk. If flag is true, the data is saved; else the data is not saved. The two conditional statements provided by Java are: `if … else` and `switch-case`.

if…else

The syntax of `if` statement is as follows:

```
if (x = = 0)
     {// Lines of code}
else if(x = = 1)
     {// Lines of code}
.....
else
     {// Lines of code}
```

The arguments to a conditional statement like `if` must be a boolean value, which is something that evaluates to true or false. You can have *n* number of `else if (){}` statements in your program, as per your requirement. The `if...else` condition can also be nested as shown.

```
if (condition)
{
     if (condition)
          {//do something based on the condition}
}
```

The following example shows how `if...else` conditional statements can be used in Java.

Example 3.8 **if...else**

```
class IFElseExample
{
    public static void main(String args[])
    {
        int x=20,y=18,z=22;
L1      if (x < y)                    // x comes before y
L2      {
L3      if (z < x)                    // z comes first
L4      System.out.println( z + " " + x + " " + y);
L5      else if (z > y)               // z comes last
L6      System.out.println(x + " " + y + " " + z);
L7      else                          // z is in the middle
L8        System.out.println(x + " " + z + " " + y);
L9      }
L10     else
```

```
L11          {                              // y comes before x
L12          if (z < y)                     // z comes first
L13          System.out.println(z + " " + y + " " + x);
L14          else if ( z > x)               // z comes last
L15          System.out.println(y + " " + x + " " + z);
L16          else                           // z is in the middle
L17          System.out.println(y + " " + z + " " + x);
           }
       }
   }
```

Output

```
C:\>Java IFElseExample
18 20 22
```

Explanation

L1 It shows `if` statement comparing `x` with `y`. If `x` is less than `y`, then control passes into the enclosing curly brackets starting from L2. But in our example, `x` is greater than `y`, so the control passes to the `else` statement in L10.

L3 It uses the nested `if` statement. This `if` clause is within the `if` statement on L1. If condition in L1 returns true, then the condition on this line is checked. The condition checks whether `z` is less than `x`, which is already less than `y` from L1. If (`z < x`) is true, then `z` is the smallest of the three, `y` is the largest, and `x` lies in between.

L4 It prints the facts of L3.

L5 If condition on L3 returns false, the control passes on to L5, which means `z` is not less than `x` and in L5, `z` is compared with `y`. If `z` is greater than `y`, it means `z` is the largest, `x` is smallest, and `y` lies in between. (We already know the fact from L1 that `x` is less than `y`).

L6 It prints the facts of L5.

L7 If condition on L5 returns false, then the control passes on to the `else` on L7, which means that `x` is less than `y` (L1) and `z` is not less than `x` (L3) and is not greater than `y` (L5). So `x` is the smallest, `y` is the largest, and `z` lies in between.

L8 It prints the facts of L7.

L10 The `else` of `if` on L1. The control passes on to this `else` if L1 returns false, which means `x` is not less than `y`.

L11 The starting curly bracket of `else`.

L12 It checks if `z` is less than `y`. If true, `z` comes first, then `y`, and `x` is the largest.

L13 It prints the facts of L12.

L14 It checks if `z` is greater than `x`. If true, `y` comes first, then `x`, and `z` is the largest. In our example, this case is executed as the value of `x` is 20, `y` is 18, and `z` is 22.

L15 It prints the facts of L14.

L16 If `z` is not less than `y` (L12) and `z` is not greater than `x`, i.e., `z` is in the middle, `y` is the smallest, and `x` is the largest.

L17 It prints the facts of L16.

Switch-case

Java has a shorthand for multiple `if` statement—the `switch-case` statement. Here is how we can write the above program using a `switch-case`:

```
switch (x) {

case 0:
     // Lines of code
```

```
        doSomething0();
        break;

    case 1:
        // Lines of code
        doSomething1();
        break;
        . . .
    case n:
        // Lines of code
        doSomethingN();
        break;

    default:
        doSomethingElse();
    }
```

`switch-case` works with `byte`, `short`, `char`, and `int` primitive type. It can also be an `enum` type (see Chapter 6) or one of the four special wrapper classes (see Chapter 6) namely: Byte for `byte`, Short for `short`, Character for `char`, Integer for `int`. We can use strings also with the `switch-case` from Java 7 onwards. It means that `x` must be one of these `int`, `byte`, `short`, `char`, `enum type`, `String` or (one of the four) `wrapper classes`. It can also be an expression that returns an `int`, `byte`, `short`, `char or String`. The value in `x` is compared with the value of each `case` statement until one matches. If no matching case is found, the default case is executed.

Once a case is matched, all subsequent statements are executed till the end of the `switch` block or you break out of the block. Therefore, it is common to include the `break` statement at the end of each `case` block, unless you explicitly want all subsequent statements to be executed. The following example shows how `switch-case` can be used in Java. A `switch-case` is more efficient than an `if-then-else` statement, as it produces a much efficient byte code.

Example 3.9 **switch-case**

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

L1    char c='B';
L2    switch(c)
L3    {
L4    case 'A':
L5        System.out.println("You entered Sunday");
L6        break;
L7    case 'B':
          System.out.println("You entered Monday");
          break;
L8    case 'C':
          System.out.println("You entered Tuesday");
          break;
L9    case 'D':
```

```
            System.out.println("You entered Wednesday");
            break;
L10     case 'E':
            System.out.println("You entered Thursday");
            break;
L11     case 'F':
            System.out.println("You entered Friday");
            break;
L12     case 'G':
            System.out.println("You entered Saturday");
            break;
L13         default:
L14                 System.out.println("Wrong choice");
L15         }
L16     }
    }
```

Output

```
You entered Monday
```

Explanation

L1 It declares a character variable `c` with the value 'B'. \

L2 It switches the control the case where a match was found. In our case, the control passes to L7.

L3 It is the start of `switch` statement.

L4–6 These show the first case, that is, case 'A'. If the value in the character variable is 'A', then this case is executed, and the output will be `You entered Sunday`. L6 shows the `break` statement, to break out of the `switch-case` statement. If the `break` statement is not included in the code, then subsequent cases will also be executed.

L7 It shows the second case similar to L4. In our example, the value of the `char` variable is 'B', so L2 switches control to this line and the output will be `You entered Monday`. After printing the output, the control moves out of the `switch-case` because a `break` statement is included in the case.

L8–12 These are similar to L7 but will only be executed in case the value of the `char` variable is 'C' (L8), 'D' (L9), 'E' (L10), 'F' (L11), and 'G' (L12), respectively.

L13 It shows the default case. It is executed in case the `char` variable takes a value other than A to G.

Note The value of character 'c' is fixed as 'B' in our example. This value should be set based on the user's input. But taking user's input is not yet discussed; so we have fixed the value of character 'c'. We will discuss it in Chapter 9.

3.9.2 Loops

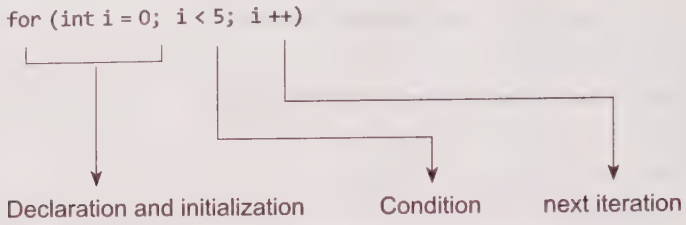
The purpose of loop statements is to execute Java statements many times. There are three types of loops in Java—`for`, `while`, and `do-while`.

`for` Loop

The `for` loop groups the following three common parts together into one statement:

(a) Initialization
(b) Condition
(c) Increment or decrement

To execute a code for a known number of times, `for` loop is the right choice. The syntax of `for` loop is



Example 3.10 **for loop**

```
class ForDemo
{
   public static void main(String args[])
   {
L1   for(int i = 1;i <= 5;i++)
L2   System.out.println("Square of "+i+" is "+ (i*i));
   }
}
```

Output

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
```

Explanation

L1 It shows the `for` loop with its three parts: initialization (`i` is initialized to 1), condition (`i` is less than or equal to 5) and the third is increment (`i++`). The loop will be executed five times. This loop has only one statement. Initially the value of `i` will be 1, for which the print statement in L2 will be executed and likewise for `i` = 2, 3, 4, and 5.

L2 It prints the square of `i` (i.e., `i*i`). This line will be executed five times, once for each value of `i`.

`while` Loop

The `while` loop is used to repeatedly execute a block of statements based on a condition. The condition will be evaluated before the iteration starts. A `for` loop is useful when you know the exact number of iterations. If you want to execute some statements for an indefinite number of times (i.e., number of iterations is unknown), a `while` loop may be the better choice. For example, if you execute a query to fetch data from a database, you will not know the exact numbers of records (rows or columns) returned by the query. A `for` loop cannot be used to iterate the returned records in this case.

The `while` statement has the following syntax:

```
while (condition)
{
      Statements to execute while the condition is true
}
```

The program in Example 3.10 can also be written using a `while` loop.

Example 3.11 **`while` Loop**

```
class WhileDemo
{
   public static void main(String args[])
   {
L1      int i = 1;
L2      while(i <= 5)
L3      {
L4      System.out.println("Square of " +i+ "is" +(i*i));
L5      i++;
L6      }
   }
}
```

Output

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
```

Explanation

L1 It initializes an `int` variable i to 1.

L2 It demonstrates the `while` loop. In this loop, the value of `i` is checked to be less than or equal to 5, which in the first iteration is 1 (less than 5), so the control passes into the loop. After executing the statements within the enclosing curly brackets of the `while` loop, again the condition of the `while` loop is checked. It goes on until the condition in the `while` returns false (i.e., when value of `i` becomes 6), in which case the control comes out of the loop.

L3 Curly bracket to denote the start of `while` loop.

L4 It prints the square of `i`.

L5 It increments the value of `i`.

L6 Curly bracket denoting the end of `while` loop.

`do-while` Loop

A `do-while` loop is also used to repeatedly execute (iterate) a block of statements. But, in a `do-while` loop the condition is evaluated at the end of the iteration. So the `do-while` loop (unlike the `while` loop) will execute at least once and after that depending upon the condition.

The general form of a `do-while` loop is

```
do
{
      Statements to execute once and thereafter while the condition is true
      } while (test);
      Next-statement;
```

Example 3.12 **do-while Loop**

```
class DoWhileDemo
{
     public static void main(String args[])
L1       { int i = 1;
L2       do
L3       {
L4       System.out.println("Square of" +i+ "is" + (i*i));
L5       i++;
L6       }while(i <= 5);
       }
     }
```

Output

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
```

Explanation

L1 It initializes an `int` variable `i` to 1.

L2 It shows the starting `do` statement of the `do-while` loop.

L4 to 6 The statement on L4 and L5 will be executed at least once and later it depends on the condition specified in the `while` loop on L6. In this case, the statements on L4 and L5 will be executed not only once, but five times (value of `i` loops from 1 to 5 inclusive).

`for-each` Loop

Java 5 introduced what is sometimes called a `for-each` statement that accesses each successive element of an array, list, or set without being associated with iterators or indexing. This new `for` statement is called the `enhanced for` or `for-each`. This loop is used to access each value successively in a collection of values (like array). It is commonly used to iterate over an array or a collections class (e.g., `ArrayList`). Like `for` loops, these loops perform a fixed number of iterations. But unlike them, the `for-each` loop determines its number of steps from the size of the collection.

The general form of `for-each` loop is

```
for (type var : arr)
{
     // Statements to repeat
}
```

We will return to `for-each` loop when we discuss arrays and collections in Java.

3.9.3 Branching Mechanism

Java does not offer a `go to` type of statement as in some older languages, because it leads to unreadable code. However, Java supports other ways to jump from one statement to another. Two types of branching statements are available in Java—*break* and *continue*.

break Statement

`break` statement is used in case the user needs to jump out of a loop, while the `continue` statement is used where the user wants to go back to the top of the loop. A `break` statement is used to jump out of a loop when a particular condition occurs, as shown below:

```
while (i < 5) {
      //do Something;
      if(i < 0) break; // jump out of the loop
}
```

The `break` will result in the program flow moving out of the loop to the next statement following the loop statement. The following example is a program statement to choose prime numbers within a given range.

Example 3.13 **Usage of break**

```
class PrimeDemo{
public static void main(String[] args){
       int j,k;
       System.out.print("Prime numbers between 1 to 30 : ");
L1     for (j = 1; j < 30; j++){
L2     for (k = 2; k < j; k++){
L3       if(j % k == 0) break;
         }
L4               if(j == k) {
L5               System.out.print(j+ " ");
       }
     }
   }}
```

Output

```
C:\Javabook\programs\chap3>Java PrimeDemo
Prime numbers between 1 to 30 : 2 3 5 7 11 13 17 19 23 29
```

Explanation

L1 It creates a `for` loop which ranges from 1 to 30 (as we need to find primes between 1 and 30).

L2 It creates an inner `for` loop which starts from 2 (as 1 is not a prime number) to `j`.

L3–5 Condition to check whether `j` is divisible by any number in the range 2 to `j-1`. If it is divisible by any number in this range (i.e., remainder is 0), `break` out of the inner `for` loop and check (in L4) whether the numerator (`j`) and denominator (`k`) are same. (Prime numbers are divisible by 1 and itself). If both are same, it is a prime number.

If, instead, you want the flow to jump out of both the loops, use the `labeled break` as shown in the next example.

Example 3.14 **Labeled break**

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

```
    {
L1  Outer : for(int i = 0; i < 4; i++){
L2      for(int j = 1; j < 4; j++){
L3      System.out.println("i:" + i + " j:" + j);
L4      if(i == 2) break Outer;
    }}}}
```

Output

```
C:\Javabook\programs\chap3>Java LabeledBreakDemo
i:0 j:1
i:0 j:2
i:0 j:3
i:1 j:1
i:1 j:2
i:1 j:3
i:2 j:1
```

Explanation

L1 A label named `Outer` is placed on the outer `for` loop with a colon after the label name.

L2 An inner `for` loop is created.

L3 Prints the value of `i` and `j`.

L4 If the value of `i` is equal to 2, the control comes out of both the loops and the program terminates.

Continue Statement

Situations can occur where you do not want to jump out of a loop, but simply stop the current iteration and go back to the top and continue with the next iteration, as shown in the following code.

Example 3.15 Code Snippet for `continue`

```
L1      while (i < 5){
L2      //doSomething1;
L3      if(i < 4) continue;
L4      //doSomething2;
    }
```

Explanation

L1 Beginning of `while` loop.

L2 Inside the loop, there are some statements shown in comments (`//do something1`).

L3 If `i` is less than 4, `continue` to the top of the loop for next iteration.

L4 The `doSomething2` statement will not execute until `i` equals 4 because the `continue` statement keeps sending the program flow back to the next iteration of the loop.

Sometimes you may want to jump out of not only the inner loop but the outer loop as well. In that case, you can put a label (similar to label in break) on the outer loop and jump to it and continue its next iteration, as in the following example.

Example 3.16 Code Snippet for Labeled `continue`

```
L1      jmp0: while (i < 5){
L2      for (int i = 0; i < 4; i++){
L3      if(i = = 2) continue jmp0; //do something;
        }
        }
```

Explanation

L1 Labelled `continue` (i.e., `jmp0`) on the beginning of `while` loop.
L2 Inner `for` loop.
L3 The `if` statement inside the inner `for` loop states to jump to the outer `while` loop if `i` is equal to 2, else execute the statements (do something).

SUMMARY

Java is an object-oriented programming language that can be used to solve problems. All the Java keywords have a fixed meaning and form the building block for program statements.

Variables hold data at memory locations allocated to them. There are eight basic data types in Java, namely byte, short, int, long, float, double, boolean, and char. Java is portable across computer platforms. Java does not leave the size of data types to the machine and the compiler, but specifies everything. All integer (byte, short, int, long) and floating-point types (float, double) are signed in Java. Java 7 introduced binary literals to be assigned to numeric variables and underscores to be used with literals. Apart from this, Java 7 added strings to be used with switch case statements.

There are several operators in Java that can be classified as arithmetic, relational, logical, assignment, increment and decrement, conditional, bit-wise, and special. Expressions are formed with variables and operators. Operators in Java have certain precedence and associativity rules that are followed while evaluating expressions. Automatic-type conversion takes place according to a set of rules in expressions with mixed types. Explicit type conversion (casting) is also possible in Java.

EXERCISES

Objective Questions

1. In the following class definition, which is the first line (if any) that causes a compilation error? Select the correct answer.

```
public class CastTest {
 public static void main(String args[]){
   char a;
   int j;
   a = 'A';        //1
   j = a;          //2
   a = j + 1;      //3
   a++;            //4
 }
}
```

   (a) The line labelled 1.
   (b) The line labelled 2.
   (c) The line labelled 3.
   (d) The line labelled 4.

2. Which of these assignments are valid?
   (a) short s = 48; (b) float f = 4.3;
   (c) double d = 4.3; (d) int I = '1';

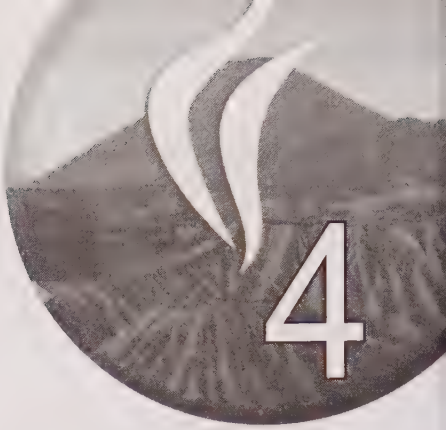
3. What is the output when the following program is compiled and run?

```
class test {
 public static void main(String args[]){
   int i,j,k,l=0;
   k = l++;
   j = ++k;
   i = j++;
   System.out.println(i);
 } }
```

   (a) 0 (b) 1 (c) 2 (d) 3

4. What gets printed on the standard output when the following class is compiled and executed? Select the correct answer.

```
public class SCkt {
 public static void main(String args[]) {
   int i = 0;
   boolean t = true;
   boolean f = false, b;
   b = (t && ((i++) == 0));
```



Classes and Objects

Space is big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist's, but that's just peanuts to space.
Douglas Adams

After reading this chapter, the readers will be able to

- know how classes and objects are created and applied in Java
- know how methods are created and used
- understand the concepts of polymorphism and overloading
- understand what is a constructor
- establish familiarity with static keyword
- know about arrays and command-line arguments
- understand inner classes
- understand and use arrays

4.1 CLASSES

Java is an object-oriented language. In the first chapter, we have learnt the concepts of object-oriented programming (OOP). Before applying these concepts in Java, we must understand the basic building blocks of OOP, i.e., *classes* and *objects*.

In the real physical world, everyday we come across various objects of the same kind. One of the many things we come across are motorbikes. In terms of object-oriented language, we can say that the bike object is one instance of a class of objects known as 'motorbikes.' Bikes have gears, brakes, wheels, etc. They also follow certain behaviors, when functions are applied on them, e.g., bikes slow down when brakes are applied, they accelerate when geared up and acceleration is applied, and so on.

Manufacturers produce many bikes from the same blueprint by taking advantage of the fact that bikes share similar characteristics. It would be very inefficient to produce a new blueprint for every individual bike they manufactured.

In the object-oriented software, there are many objects of the same kind, i.e., belonging to the same classes that share certain characteristics. Like the bike manufacturer, we can take advantage of the fact that objects of the same kind are similar and a blueprint for those objects can be created. Software 'blueprints' for objects are called *classes*.

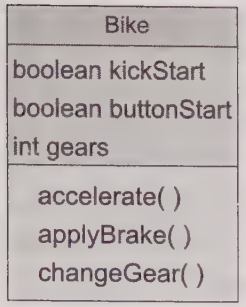


Fig. 4.1 Bike Class

Let us come back to our bike class, which would also declare and provide implementations for the instance methods or functions that allow the rider to change gears, apply brakes, and accelerate. Figure. 4.1 shows the bike class.

Note A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind. In other words, a class can be thought of as a user-defined data type and an object as a variable of that data type that can contain data and methods, i.e., functions working on that data.

4.2 OBJECTS

The object-oriented technology revolves around objects. We see many objects around us such as table, chair, dog, fan, computer, pen, and car. These objects need not be tangible ones only, but can be intangible also, e.g., bank accounts, marks, fees, etc. All these real-world objects have different *states* and *behaviors*. The state of an object is defined by the values of the attributes at any instant. Bikes have attributes (speed, engine capacity, number of wheels, number of gears, brakes), behaviors (braking, accelerating, slowing down, and changing gears) and on application of this behavior on attributes, the state of the object will change. Bike object can be in various states, it can be stationary, moving etc. For example, when we apply brakes, the speed will reduce and when we accelerate speed increases. A state will change over time and at any instant a state would be somewhat like current speed = 60 km/hr and current gear = 4th. Similarly, the state of a fan would be either off or on.

We can conceptualize these real-time objects as software objects. They are similar natured in the sense they too have states and behaviors. The state in software objects is maintained in variables and the behavior can be implemented using methods. It is interesting to know that these real-world objects can be represented using software objects.

Note An object is a software bundle that encapsulates variables and methods operating on those variables.

You might want to represent a real-world bike as a software object in a gaming application. Abstract objects representing abstract concepts can also be modeled using software objects. For example, a *bank account* is a common object used in banking solutions to represent the details of bank accounts of various customers of a bank.

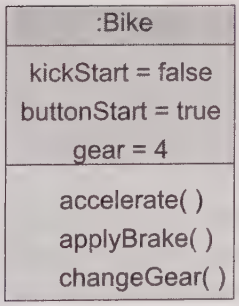


Fig. 4.2 Bike Object

Figure. 4.2 shows a common visual representation of a software object.

It would be correct to say that everything the software object knows (state) and can do (behavior) is expressed by the variables and methods within that object. A software object that models the real-world bike would have variables that indicate the bike's current state: its speed is 10 mph, its acceleration in terms of revolutions per minute is 5000 rpm, and its current gear is 4th. These variables are known as *instance variables*.

The object *bike* would also have methods to brake, accelerate, and change gears. These are known as *instance methods*. Only relevant fields and behaviors are added into a class. For example, a bike does not have a surname, and it cannot speak or sleep. A bike class can be created that declares several instance variables to contain the gears, the brakes, and so on, for each bike object and every bike will have its own brakes, gears, etc.

It is worth noting here that all object instances have their own copies of instance variables. This means that if there are five object instances of a bike class, there are five copies of each instance variable defined in that class. Each object has its own copy of instance variables which is different from other objects created out of the same class.

The values of the instance variables are provided by each instance of the class. So, after you have created the bike class, you must *instantiate* it (create an object of it) before you can use it. When an instance of a class is created, an object of that type is created and memory is allocated by the system for the instance variables declared by the class. Then the object's instance methods can be invoked to perform operations.

Note Instances of the same class share the instance method implementations (method implementations are not duplicated on a per object basis).

In addition to instance variables and methods, classes can define their own *class variables* and *methods*. Every object will have its own *instance variables* but *class variables* will be shared by all the objects of the class. You can access *class variables* and *methods* using an instance of the class or using the class name. You need not instantiate a class to use its *class variables* and *methods*. Class methods can only access the class variables directly. They don't have direct access to instance variables or methods. A single copy of all *class variables* is created and all instances of that class share it. For example, suppose all cars had the same number of gears. In such a situation, a class variable can be created that defines the number of gears. All instances of the class will share this variable. If any object manipulates the class variable, then it changes for all objects of that class.

4.2.1 Difference Between Objects and Classes

Both objects and classes look the same. Yes, it is a fact that the difference between classes and objects is often the source of some confusion. In the real world, it is obvious that classes are not themselves the objects that they describe—a blueprint of a bike is not a bike. However, it is difficult to differentiate between classes and objects in programming. This is partially because objects in programming are merely the electronic models of real-world objects or abstract concepts. Classes have logical existence, whereas objects have physical existence, e.g., furniture itself does not have any physical existence, but chairs, tables, etc. do have.

4.2.2 Why Should we Use Objects and Classes?

Modularity, information hiding, i.e., *data encapsulation*, can be incorporated using objects. Classes, being blueprints, also provide the benefit of reusability along with the ease of changing and debugging code. For example, bike manufacturers reuse the same blueprint over and over again to build lots of bikes. Programmers use the same class repeatedly to create many objects.

4.3 CLASS DECLARATION IN JAVA

Declaring a class is simple. A class can be declared using the keyword `class` followed by the name of the class that you want to define. Giving a name to a class is something which is totally in the hands of the programmer. But while doing so, he must take care of the relevance of the class name, the legality of Java identifiers used as the class name, and the naming convention used in Java. Thus, the simplest class declaration looks as follows:

```
class Bike
{
   //Variables declaration
   //Methods declaration
}
```

Example 4.1 Class Declaration

```
class GoodbyeWorld
{
public static void main (String args[])
     {
     System.out.println("Goodbye World!");
     }
}
```

Here the name of the class is `GoodbyeWorld`. The class just contains the `main()` method, which is responsible for displaying `GoodbyeWorld` on the screen.

To sum up, all the action in a Java program takes place inside the class. Methods and variables are defined inside the classes. The class is the fundamental unit of programming in Java. The class declaration can specify more about the class, like you can:

- declare the superclass of a class
- list the interfaces implemented by the class
- declare whether the class is `public`, `abstract`, or `final`

For each of the cases above, the class declaration will differ accordingly. We will talk about that as and when we cover the related concepts. Taking all the possibilities of class declaration in Java, we can summarize the class declaration syntax as

```
[modifiers] class ClassName [extends SuperClassName] [implements InterfaceNames]
{ . . . }
```

The items enclosed inside [ ] are optional. A class declaration defines the following aspects of the class:

- `modifiers` declare whether the class is `public`, `protected`, `default`, `abstract` or `final`
- `ClassName` sets the name of the class you are declaring
- `SuperClassName` is the name of the `ClassName's` superclass
- `InterfaceNames` is a comma-delimited list of the interfaces implemented by `ClassName`

Only the `class` keyword and the class name are mandatory. Other parameters are optional.

Note The Java compiler assumes the class to be non-final, non-public, non-abstract, subclass of objects (discussed in Chapter 6) that implements no interfaces if no explicit declaration is specified.

Certain terms in the above syntax such as `modifiers`, `extending superclasses`, and `implementing interfaces`, which are presently unfamiliar, will be discussed in the later chapters.

4.3.1 Class Body

The class contains two different sections: variable declarations and method declarations. The variables of a class describe its state, and methods describe its behavior. All the member variables and methods are declared within the class. There are three types of variables in Java: *local variables*, *instance variables*, and *class variables*. Local variables are defined inside a method. Instance variable is defined inside the class but outside the methods, and class variables are declared with the static modifier inside the class and outside the methods. For now, we will concentrate on instance variables.

Instance Variables

It is important to understand that a class can have many instances (i.e., objects) and each instance will have its own set of instance variables. Any change made in a variable of one particular instance will not have any effect on the variables of another instance. For more details on class variables, local variables, and instance variables, see Section 4.7.

Normally, you declare the member variables first followed by the method declarations and implementations.

```
classDeclaration
{
   memberVariableDeclarations
   methodDeclarations
}
```

Let us see how you can declare instance variables in a class. Example 4.2 shows a sample class declaration with two instance variables. We will return to the discussion of instance variables later in the chapter. Please note that if you try to run this example it won't show any output because it is not fully functional and there are certain statements/methods that we need to add so that this program can display any output, which will follow later in the chapter.

Example 4.2 Class Declaration

```
L1    class SalesTaxCalculator {
L2       float amount = 100.0f;    // instance variable
L3       float taxRate = 10.2f;    // instance variable
L4    }
```

Explanation

L1 Class declared with the keyword `class` followed by the name of the class `SalesTaxCalculator`.

L2 A instance variable *amount* is declared to denote the amount on which sales tax has to be calculated.

L3 Declares another float instance variable `taxRate` to denote the rate of tax on the sale amount.

L4 End of the class.

The above example shows a class with two instance variables. Instance variables are part of the instance of the class (object). These instance variables will be created when the instance is created. In order to be able to access/manipulate these instance variables, we need to create objects of this class. We have already seen what objects are. Let us see how objects are created and used in Java.

4.4 CREATING OBJECTS

In Java, you create an object by creating an instance of a class or, in other words, instantiating a class. A Java object is defined as an instance of a class. The type of the object is the class itself. Often, you will see a Java object created with a statement like

```
SalesTaxCalculator obj1 = new SalesTaxCalculator();
```

This statement creates a new `SalesTaxCalculator` object. This single statement declares, instantiates, and initializes the object. `SalesTaxCalculator obj1` is a reference variable declaration which simply declares to the compiler that the variable `obj1` will be used to refer to an object whose type is `SalesTaxCalculator`. The new operator instantiates the `SalesTaxCalculator` class (thereby allocating memory and creating a new `SalesTaxCalculator` object), and `SalesTaxCalculator()` initializes the object.

4.4.1 Declaring an Object

Object declarations are same as variable declarations. For example,

```
SalesTaxCalculator obj1;
```

Generally, the declaration is as follows:

```
type name
```

where `type` is the type of the object (i.e., class name) and `name` is the name of the *reference variable* used to refer the object. Classes are like new data types. So `type` can be any class such as the `SalesTaxCalculator` class or the name of an *interface*.

Note A variable holds a single type of literal, i.e., 1, bat, 345, etc. An object is defined as an instance of a class with a set of instance variables and methods that perform certain tasks depending on what methods have been defined for. A reference variable is used to refer/access an object. A reference variable is of a specific type name of the class is its type. Unlike normal variable, reference variables can be static, instance or local variables as well as they can be passed to or returned from the method.

The above declaration won't create an object. It will create a variable with a name and specify its type. For example, `SalesTaxCalculator` is the type and `obj1` is the reference variable.

4.4.2 Instantiating an Object

After declaring a variable to refer to an object, an actual, physical copy of the object must be acquired and assigned to that variable. This can be achieved by the `new` operator. The `new` operator instantiates a class by dynamically allocating (i.e., at runtime) memory for an object of the class

type and returns a reference to it. This reference is nothing but the address in the memory of the object allocated by `new`. This reference or memory address is then stored in the variable declared. The `new` operator requires a single argument, i.e., a `constructor` call. The `new` operator creates the object or instantiates an object and the constructor initializes it.

```
SalesTaxCalculator obj1 = new SalesTaxCalculator()
```

The above statement just creates an instance of a class, `SalesTaxCalculator`. In other words, the `new` operator creates an object `obj1` by allocating memory for its member variables, i.e., `amount` and `taxRate` (Example 4.2) and few other items.

4.4.3 Initializing an Object

By initializing an object, we mean the instance variables are assigned initial values. The instance variables of a particular object will have different values during the lifetime of an object. But to start with, initial values are required. If no value is specified for the instance variables, then the default values will be assigned to those variables based on their respective types. Initial values can be provided by *instance variable initializers* and *constructors*.

Instance variable initializers are values directly assigned to the instance variable outside any method/constructor but within the class. [As shown in L2 and L3 of Example 4.2(a)].

The best and convenient approach is to create your own constructor. Constructors should be provided within classes to initialize objects. Constructors have the same name as that of the class. Constructors are invoked as soon as the object is created. In case you do not create a constructor for your class, Java compiler provides a default constructor for your class automatically. The default constructor is a zero argument constructor with an empty body. The implicitly created default constructor is invoked as soon as the object is instantiated with `new` keyword as shown below.

```
new SalesTaxCalculator()
```

We will come back to the concepts of constructors along with the examples explaining them in Section 4.6.

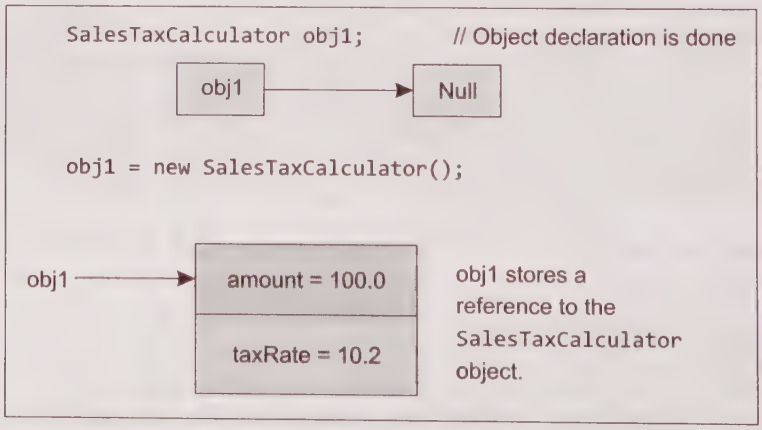


Fig. 4.3 Steps in Object Creation

To sum up, the final object creation can be said as complete, when the objects are initialized, either with an implicit constructor or an explicit constructor. This object creation can be used in a programming code in two ways:

```
SalesTaxCalculator obj1 = new SalesTaxCalculator();
```

Here all the three operations, object declaration, object instantiation, and object initialization are done by a single statement. The above process takes place in the following way:

Now that we know how to create a class and objects for that class, we can rewrite Example 4.2 where we can do these things in one program only. The following program displays a class `SalesTaxCalculator`, with two instance variable (initialized to some values) and two objects of the class `SalesTaxCalculator`, `obj1` and `obj2` (created inside the `main` method). Instance variable initializers are used in this example to initialize objects: `obj1` and `obj2` (Fig. 4.3).

Example 4.2 (a) Object and Classes

```
L1    class SalesTaxCalculator {
      // instance variable initializer
L2    float amount = 100.0f;
      // instance variable initializer
L3    float taxRate = 10.2f; //instance variable
      // instance method
L4    public static void main (String args[ ])
      {
L5       SalesTaxCalculator obj1 = new SalesTaxCalculator();
L6       SalesTaxCalculator obj2 = new SalesTaxCalculator();
L7       System.out.println("Amount in Object 1: "+ obj1.amount);
L8       System.out.println("Tax Rate in Object 1: "+ obj1.taxRate);
L9       System.out.println("Amount in Object 2: "+ obj2.amount);
L10      System.out.println("Tax Rate in Object 2: "+ obj2.taxRate);
      }}
```

Output

```
D:\javabook\programs\chap4\java SalesTaxCalculator
Amount in Object 1: 100.0
Tax Rate in Object 1: 10.2
Amount in Object 2: 100.0
Tax Rate in Object 2: 10.2
```

Explanation

L1 Class declaration.

L2 & L3 Instance variable have been declared with their initializer.

L4 Main method declared.

L5 & L6 Two objects of this class are created in these lines. As already discussed, the `new` keyword allocates memory to these objects according to the size of instance variables (plus a few more bits for some more items). Note that no constructor has been created in this class, so the Java compiler will automatically provide a default constructor for this class which is being invoked while creating object in these

statements. The default constructor is empty, so it is the responsibility of the Java compiler to ensure that the instance variable are initialized to their respective values (`amount = 100.0f` and `taxRate = 10.2f`) according to their initializer (mentioned in L2 and L3) by the JVM at runtime. (How does it ensure this? We will discuss it later in the chapter). For instance, we will consider that the instance variable initializers are used by the Java compiler to initialize these instance variables.

L7 Is a print statement that prints the value of the instance variable present in `obj1`. The variable can be accessed through the object followed by the dot operator from main method or outside the class (depends on access and scope of the object).`obj1.amount` will return the value of amount stored in the instance `obj1`. The value of the variable can be changed by using the following syntax:

```
obj1.amount=200.0f;
```

L8, 9 & 10 Similar to L7.

Note Setter methods can also be used for assigning or modifying values of instance variables (i.e., `set X()` or `set Y()` where `x` and `y` are the names of the instance variables) They are declared inside a class, as shown in Examples 4.3 and 4.5 (methods will be discussed in the next section).

The above program has a limitation that all objects created will have the same value for `amount` and `taxRate`. Later on it can be changed using object references. But it would be wiser to let all objects have their own different amount and tax rates as soon as they are created. This problem will be solved using constructors (Section 4.6).

4.5 METHODS

The word *method* is commonly used in object-oriented programming. It is similar to a function in any other programming language. Many programmers use different terms, especially *function*, but we will stick to the term *methods*. None of the methods can be declared outside the class. All methods have a name that starts with a lowercase character.

4.5.1 Why Use Methods?

- **To Make the Code Reusable** If you need to do the same thing or almost the same thing, many times, write a method to do it and then call the method each time you have to do that task.
- **To Parameterize the Code** You will often use parameters to change the way the method behaves.
- **For Top-down Programming** You can easily solve a bigger problem or a complex one (the 'top') by breaking it into smaller parts. For the same purpose, we write methods. The entire complex problem (functionality) can be broken down into methods.
- **To Simplify the Code** Because the complex code inside a method is hidden from other parts of the program, it prevents accidental errors or confusion.

4.5.2 Method Type

There are two types of methods in Java: *instance methods* and *class methods*. Instance methods are used to access/manipulate the instance variables but can also access class variables. Class

methods can access/manipulate class variables but cannot access the instance variables unless and until they use an object for that purpose.

4.5.3 Method Declaration

The combined name and parameter list for each method in a class must be unique. The uniqueness of a parameter is decided based on the number of parameters as well as the order of the parameters. So,

```
int methodOne (int x, String y)
```

is unique from

```
int methodOne (String y, int x).
```

Let us take a look at the general syntax of a method declaration:

```
[modifiers] return_typemethod_name (parameter_list)
  [throws_clause] {
     [statement_list]
  }
```

The parameters enclosed within square brackets [ ] are optional. The square brackets are not a part of the code; they are included here to indicate optional items. We will discuss only those parts that are required at the moment and leave the rest for the later chapters. The method declaration includes

- **Modifiers** If you see the syntax of the method declaration carefully, there is an optional part of it, *modifiers*. There are a number of *modifiers* (optional) that can be used with a method declaration. They are listed in Table 4.1.

Table 4.1 Optional Modifiers used While Declaring Methods

Modifier	Description
`public, protected, default or private`	Can be one of these values. Defines the scope—what class can invoke which method?
`static`	Used for declaring class methods and variables. The method can be invoked on the class without creating an instance of the class.
`abstract`	The class must be extended and the abstract method must be overridden in the subclass.
`final`	The method cannot be overridden in a subclass.
`native`	The method is implemented in another language (out of the scope of this book).
`synchronized`	The method requires that a monitor (lock) be obtained by calling the code before the method is executed.
`throws`	A list of exceptions is thrown from this method.

- **Return Type** It can be either `void` (if no value is returned) or if a value is returned, it can be either a primitive type or a class. If the method declares a return type, then before it exits, it must have a `return` statement.
- **Method Name** The method name must be a valid Java identifier. We have already discussed Java identifiers in Section 3.3.
- **Parameter List** Zero or more type/identifier pairs make up a parameter list. Each parameter in the parameter list is separated by a comma.
- **Curly Braces** The method body is contained in a set of curly braces. Methods contain a sequence of statements that are executed sequentially. The method body can also be empty.

In Example 4.2(a), we have stored data: *amount* and *taxRate* but we have not calculated the tax amount based on the rate. We need to calculate the tax amount. This operation would require some calculation (operations) to be performed on the data variables. These operations will be performed inside a method so we need to add a method to that class and revise the class. The method added is `calculateTax()` (L4) which calculates the taxed amount. This method is invoked in L10 and 11 by the two objects using the dot operator. Note that the answer in both cases is the same because the data in both cases is same, i.e., amount and tax rate are same for both objects so the taxed amount is same.

Example 4.2 (b) Added Instance Method

```
L1    class SalesTaxCalculator {
      // instance variable initializer
L2    float amount = 100.0f;
      // instance variable initializer
L3    float taxRate = 10.2f;              //instance variable
      // instance method
L4    void calculateTax() {
L5          float taxAmt = amount*taxRate/100;
L6    System.out.println("The Taxed Amount is: "+taxAmt);
      }
L7    public static void main (String args[ ])
      {
L8          SalesTaxCalculator obj1 = new SalesTaxCalculator();
L9          SalesTaxCalculator obj2 = new SalesTaxCalculator();
L10         obj1.calculateTax();
L11         obj2.calculateTax();
      }}
```

Output

```
The Taxed Amount is: 10.2
The Taxed Amount is: 10.2
```

Let us take a different example to explain the concepts in more detail. The following example has a couple of *instance* methods, `setRadius()` and `calculateArea()`, declared inside the class, `Circle`. The word *instance* has been particularly used to distinguish between instance and class methods. The modifier `static` has not been used while declaring methods so the methods become instance methods.

Example 4.3 Instance Method Declaration

```
L1    class Circle
L2    {
L3          float pi = 3.14f;
L4          float radius;
            //setter method to change the instance variable: radius
L5          void setRadius(float rad)
L6          {
L7          radius = rad;}
L8          float calculateArea()
L9          {
L10            float area = pi * radius * radius;
L11            return (area);
L12   }     }
```

Explanation

L1 Class declaration.

L3 & 4 Instance variable declaration.

L5 Declares an instance method popularly known as setter or mutator methods (note that `static` modifier is not used in this declaration). They are known as setter or mutator methods because they set or change (mutate) the values of instance variables. The data type `void` indicates that this method will not return any value. The name of the instance method is `setRadius()`and it accepts a `float` parameter `rad`. This method is used to assign a value to the instance `variableradius`. The method argument `rad` is assigned to the instance variable `radius` in this method on L7. It also shows that instance methods can access instance variables directly. Instance methods are invoked using objects, so data residing in objects can be easily accessed (set or get) by instance methods.

L6 The body of the method starts with the left brace, "{".

L7 `rad` is assigned to an instance variable `radius` of type `float`. The right brace "}" signifies the end of the method.

L8 Declares another instance method. The instance method `calculateArea()` has been declared to return a value of type `float`.

L9 "{" signifies the start of the method body.

L10 Instance variables `pi` and `radius` are multiplied to calculate the area of the circle. As shown, instance variables can be used by instance methods directly to produce result. An important point to note is that different `Circle` objects will have different values of `radius` and obviously the calculated area will be different but the instance methods remains the same. In other words, instance methods are not implemented on a per object basis as is the case with instance variables. The area of the circle is stored in the local variable, `area`, declared as `float`.

L11 The value stored in the variable area is returned by the `return` statement.

L12 The first right brace "}" signifies the end of method, `calculateArea()`, and the second brace "}" signifies the end of class, `Circle`.

Note The responsibility of providing initial values is that of constructors, and constructors are called only once, i.e., during object creation. If the value of instance variables has to be changed, setter methods should be used and that too can be invoked any number of times.

4.5.4 Instance Method Invocation

If you run Examples 4.2, 4.2(a), and even Example 4.3, you won't see any output for a very simple reason that we have created methods but we have not invoked (called) them. Methods (instance or class methods) cannot run on their own, they need to be invoked. Instance methods will be invoked by the objects of the class they are a part of. Class methods invocation will be discussed later in this chapter.

When an object calls a method, it can pass on certain values to the methods (if methods accept them) and the methods can also return values from themselves if they wish to. Data that are passed to a method are known as *arguments* or *parameters*; the required arguments for a method are defined by a method's parameter list (method signature). Let us take an example and see how invoking is done.

Example 4.4 Instance Method Invocation

```
L1    class CallMethod
L2    {
L3    public static void main (String args[])
L4    {
L5          float area1;
L6          Circle circleobj = new Circle();
L7          circleobj.setRadius(3.0f);
L8          area1 = circleobj.calculateArea();
L9          System.out.println("Area of Circle = " + area1);
L10   }}
```

Output

```
Area of Circle = 28.26
```

Explanation

L6 Creates an object of `Circle` class. `setRadius()` and `calculateArea()` are instance methods of the class, `Circle`. So an instance is required to invoke these instance methods and that instance must be of the class the methods are a part of, i.e., `Circle` (Example 4.3). That's why an object of the `Circle` class named `circleobj` is created.

L7 & 8 Using the instance created in L6, we call the methods `setData()` and `calculateArea()` with the help of a `dot` operator. A value `3.0f` (`f` to indicate float value) is passed as an argument in the `setRadius()` method invocation. This value `3.0f` is assigned to the local float variable `rad`, which is actually an argument in the method declaration (see Example 4.3, L5). The `calculateArea()`method calculates the area and returns the value which is captured in a float variable `area1`.

The following definitions are useful in the above context.

Formal Parameter The identifier used in a method to stand for the value that is passed into the method by a caller. For example, the parameter defined for `setRadius()`, i.e., `rad` in L5 of Example 4.3 is a formal parameter, as it will be bound to the actual value sent by the caller method. These formal parameters come in the category of local variables which can be used in their respective methods only.

Actual Parameter The actual value that is passed into the method by a caller. For example, in L7 of Example 4.4, `3.0f`, passed to `setRadius()`, is the actual parameter.

Note The number and type of the actual and formal parameters should be same for a method. Also note that the class having the `main()` method is to be executed first by the *Java interpreter*.

In Example 4.3, we have created a class (`Circle`) and two methods in that class. Example 4.4 shows how the methods of `Circle` class (Example 4.3) are called from another class, i.e., `CallMethod`. The methods can also be called from within the class, as shown in Example 4.5.

Example 4.5 Adding Instance Variable(s) and Instance Method(s)

```
L1    class Circle
L2    {     float pi=3.14f;
L3          float radius;
L4    void setRadius(float rad){
L5          radius = rad;
L6    }
L7    float calculateArea(){
L8    float area = pi* radius*radius;
L9    return (area);
L10   }
L11   public static void main (String args[]) {
L12         Circle circleobj = new Circle();
L13         circleobj.setRadius(3.0f);
L14         System.out.println("Area of Circle = " + circleobj.calculateArea());
            // The above two lines can be compressed to one, i.e.
            // System.out.println(circleobj.setRadius(3.0f).calculateArea());
L15   }}
```

Explanation

The example is entirely same as that of Example 4.3 up to L10. (The output is entirely same as that of the previous program.)

L11 The execution begins at `main()`. Because `main()` is defined in this class, it can execute on its own and there is no need of a separate class like `CallMethod` (Example 4.4) for invoking the methods of the `Circle` class. The main method from that class has been squeezed out and inserted in the class `Circle` as shown in the lines 11–15.

L12 An object of the `Circle` class, named `circleobj`, is created using the `new` operator.

L13 `setRadius()`is called with the help of an object of the `Circle` class and a float argument is passed to it.

L14 `calculateArea()`is called using the object created in L12.

Note In Java, all values are passed by value. This is unlike some other programming languages that allow pointers to memory addresses to be passed into methods. When a primitive type value is passed to a method, the value is copied. The copied value, if changed inside the method, does not affect the original value. When an object is passed, only the reference is copied. There is just one object that has two references now on it. The changes made to the object through one reference will be reflected when the object is accessed through other references.

4.5.5 Method Overloading

Method overloading is one way of achieving polymorphism in Java. Each method in a class is uniquely identified by its name and parameter list. What it means is that you can have two or

more methods with the same name, but each with a different parameter list. This is a powerful feature of the Java language called *method overloading*. Overloading allows you to perform the same action on different types of inputs. In Java whenever a method is being called, first the name of the method is matched and then, the number and type of arguments passed to that method are matched.

In method overloading, two methods can have the same name but different signatures, i.e., different number or type of parameters. The concept is advantageous where similar activities are to be performed but with different input parameters. Example 4.6 shows an example of overloading a method `max()` in order to calculate the maximum value for different combinations of inputs.

Example 4.6 Method Overloading

```
class OverloadDemo
    {
L1  void max(float a, float b)
L2  {
L3     System.out.println("max method with float argument invoked");
L4     if(a > b)
L5          System.out.println(a + " is greater");
L6
       else
L7          System.out.println(b + " is greater");
    }
L8  void max(double a, double b)
    {
    System.out.println("max method with double argument invoked");
    if(a>b)
           System.out.println(a + " is greater");
    else
           System.out.println(b + " is greater");
L9  }
L10 void max(long a, long b)
    {
    System.out.println("max method with long argument invoked");
    if(a>b)
           System.out.println(a + " is greater");
    else
           System.out.println(b + " is greater");
L11 }
L12 public static void main(String args[])
L13 {
L14        OverloadDemo o = new OverloadDemo();
L15        o.max(23L,12L);
L16        o.max(2,3);
L17        o.max(54.0,35f);
L18        o.max(43f,35f);
    }}
```

Output

```
C:\javabook\programs\chap4>java OverloadDemo
max method with long argument invoked
23 is greater
max method with long argument invoked
3 is greater
max method with double argument invoked
54.0 is greater
max method with float argument invoked
43.0 is greater
```

Explanation

L1 Method `max` is defined inside class `OverloadDemo` with two arguments of type float.

L2 Marks the beginning of the method.

L3 Shows a print statement describing the method that has been invoked.

L4 `if` statement is used to check whether the float argument `a` is greater than `b`. If `a` is greater, then L5 prints `a` is greater, else L7 prints `b` is greater.

L8 & 9 Overloaded method `max` is defined in these lines. This overloaded version of the method accepts two arguments of type `double`. This is different from the `max` method defined in L1. The processing inside this method is entirely similar to the previous method with the exception that now the maximum will be chosen from two double values instead of float values.

L10 & 11 Another version of overloaded method `max` is defined in these lines. This overloaded version of the method accepts two arguments of type long. This is different from the max method defined in lines L1 and L8. The processing inside this method is entirely similar to the previous method with the exception that now the maximum will be chosen from two double values instead of float values.

L12 `main` method has been defined. Execution starts from `main` method.

L13 Marks the beginning of the `main` method.

L14 An object of the class is created to invoke the instance methods.

L15 Shows the invocation of the method, `max`, and two arguments that are passed to it. The question arises, which version of the `max` method will be invoked? (**Remember:** The invocation will be based upon the number and type of arguments). In our case, we have only two arguments in all the overloaded methods. So the decision is taken according to the type of arguments. In this particular statement, two `long` arguments are passed. First of all, Java tries to find an exact match, i.e., a method named `max` in class `OverloadDemo` which accepts two `long` arguments. Java finds the method in L10. The method is called. If an exact match could not be found (say for example, the method `max` with long arguments is not present in the `OverloadDemo` class), then Java looks for a method named `max` which has the arguments to accommodate these `long` values (**Remember:** `long` values can be accommodated implicitly only in `float` and `double`). This example has `max` methods with both `float` and `double` arguments. So which method will be called? The `max` method with `float` arguments will be called (`long` values are promoted to float and passed). And in case the `max` with `float` arguments is also not available, then the method with `double` arguments will be called (`long` values are promoted to `double` and passed).

L16 `max` method is called with two `int` arguments passed to it. In `OverloadDemo` class, Java does not find a method which accepts two `int` arguments, but it finds a method `max` that accepts two `long` arguments. These two `int` arguments are automatically promoted to `long` and passed to the method with the name `max` accepting two `long` arguments (automatic type promotion has taken place here).

L17 Shows the invocation of `max` method with a `double` argument and a `float` argument. In this case, Java does not find an exact match, as there is no such

method named `max` that accepts a `double` argument and a `float` argument. So, automatic promotion takes place in this case also. The question arises that which overloaded method will be called? The `max` method with both `float` arguments cannot be called, as the first argument that is being passed is a `double`. Similarly, the `max` method with both `long` arguments cannot be called, as both the arguments are bigger than `long`. So, the `max` method with both `double` arguments will be called, as the first argument is a perfect match and the second will be automatically promoted to `double` (see output).

L18 The `max` method with both `float` arguments will be called in this case.

Note As a general rule, automatic type promotion takes place while passing parameter values to methods. In overloading, the decision of choosing which method to invoke is resolved by the Java compiler at compile time (early-binding) rather than delaying it till runtime because

(a) Java is a strongly typed language.
(b) Resolving all these issues at compile time will avoid unnecessary exceptions at runtime.
(c) Enhanced performance.

4.6 CONSTRUCTORS

Whenever an object is created for a class, the instance variables of the class needs to be initialized, i.e., they need to be given initial values. It can be done through instance variable initializers (as shown in L2 and L3 [Examples 4.2 and 4.2(a)], L3 (Example 4.3) and L2 (Example 4.5)) and instance initialization blocks. An instance initialization block is a block of statement enclosed in parenthesis with initialization placed in it as shown below:

```
class Rectangle
{
// Instance initialization blocks
{
        length=10;
        width=10;
}
}
```

But Java has a simple and concise method of doing it. It has a mechanism for automatically initializing the values for an object, as soon as the object is created. The mechanism is to use *constructors*.

Constructors have the same name as the class they reside in and they are syntactically similar to a method. Constructors are automatically called immediately after the object for the class is created by a `new` operator. Constructors have no return type, not even `void`, as the implicit return type of a constructor is the class type itself.

In Section 4.4.3, we discussed a little about constructors, promising that we would come back to this topic. Now it is time to recall that section on object creation. An implicit or default constructor is used as a parameter to the `new` operator, just as shown below.

```
SalesTaxCalculator r1 = new SalesTaxCalculator ();
```

Here, the `new` operator is calling the `SalesTaxCalculator()` constructor. If the constructor is explicitly defined within the class just as shown in Example 4.7, it is known as *explicit constructor*, otherwise Java automatically creates a default constructor as soon as the object is instantiated by the `new` operator. They are known as *implicit* or *default* or *no-argument constructors*. In

earlier examples, no constructor was explicitly provided, so Java provided them with a default constructor. But in case you define your own constructor within the class (Example 4.7), the default constructor will not be provided by Java. In that case, the constructor defined within the class will be called.

The default constructor, provided by Java compiler, is a no-argument constructor with empty body. The only question that would arise now is that if the default constructor is an empty constructor, then how are the variables initialized to the user specific values or default values and who does it? For example in case of Examples 4.2, 4.2(a), and 4.2(b), when instance variable initializers are used and no constructors has been defined in the `SalesTaxCalculator` class, how are the objects `obj1` and `obj2` initialized with the values specified in instance variable initializers as the default constructor is an empty constructor. What happens in the background is that Java compiler creates a special method known as `<init>` method for each of the constructors specified in the class. The code explicitly written in the constructors is placed within the `<init>` method after some operations like calling the superclass constructor, instance variable initializers and instance initialization blocks in the order in which they appear in the source code. When no constructors have been specified, the Java compiler creates a default constructor and an `<init>` method for the default constructor. This method will also include a call to superclass constructor as well as the instance variable initializers and instance initialization block (if any mentioned in the class and in the order mentioned in the source code).When no constructors and no instance variable initializer or block have been specified, the Java compiler creates a default constructor and `<init>` method for the default constructor, which initializes the instance variables with their respective default values.

Note `<init>` is a special method, meant for the JVM (to initialize objects) and not the programmer. So you cannot create a method by this name in your program. Also note that the arguments of this method would be same as that of the constructors and the return type would be void. This `init` mechanism was created in Java to ensure that memory allocated is initialized properly and any bugs should not arise due to garbage values in memory as in the case of other languages like C and C++.

Table 4.2 provides a summary on constructors versus methods.

Table 4.2 Constructors vs Methods

Constructor	Methods
Do not have any return type not even `void`	Will have a return type
Will have the same name as that of class	Can have any name even the name of class (although should not be used)
Invoked as soon as the object is created and not thereafter	Invoked after the object is created (instance methods) and can be called any number of times thereafter
Constructors cannot be inherited	Methods can be inherited
Constructors can be overloaded	Methods can also be overloaded
Constructors can be private, protected, default or public	Methods can also be private, protected, default or public
Role of constructor is to initialize object	Role of method is to perform operations
Constructors cannot be abstract, final, static or synchronized	Methods can be abstract, final, static or synchronized

Let us take an example to illustrate the usage of constructor. L3 of Example 4.7 defines an explicit default constructor that does not accept any argument but initializes the instance variables to the specified values.

Example 4.7 Constructor

```
L1   class Room{
L2      double length, breadth, height, volume;
        No Argument Constructor
L3      Room(){
L1           length = 14;
L5           breadth = 12;
L6           height = 10;
L7      }

L8      // Computation of volume of the room
L9      double volComp(){
L10     volume = length * breadth * height;
L11     return volume;
L12     }
L13     public static void main (String args[ ]){
L14     Room r1 = new Room();
L15     Room r2 = new Room();
L16     System.out.println("The volume of the room is " +r1.volComp());
L17     System.out.println("The volume of the room is " +r2.volComp());
L18     }
L19  }
```

Output

```
D:\javabook\programs\chap 4>java Room
The volume of the room is 1680.0
The volume of the room is 1680.0
```

Explanation

L3 A constructor with the name of the class, `Room`, is defined. It should be noted that the constructor declaration is very much like a method declaration but does not have a return type.

L4 & 6 Various instance variables are initialized with certain values.

L9 & 12 Instance method `volComp()`is defined and implemented for calculating and returning the volume of the room to the caller. The return type of the method is specified as `double`. Return values are expected from methods when you would like to perform more operations on the returned values or want to pass them further. Here volume is returned specifically to denote how values are returned from methods. The volume calculated is stored in the instance variable `volume`, which is returned at the end of the method with the help of `return` keyword. Please note that if a method specifies a return type then it must return a value of that type using a `return` keyword.

L14 & 15 Two objects, `r1` and `r2`, are created or instantiated using `new` operator. As soon as this is done, the constructor `Room()`on L3 is called automatically, which in turn initializes all the variables that it is defined for. The default constructor will not be provided by Java because we have defined a constructor for our class. So when we create object our defined constructors will be invoked which would initialize the objects. Obviously, in the background this task will be achieved using `<init>` method.

L16 & 17 The volume of both the objects of the `room` class is printed. Note that, `volComp()`is called

by their respective objects, in order to return the value of volume. Here, the volume for both the instances will be same, because both the objects call the method `volComp()`, which uses the same set of dimensions for the volume calculation. It is so because both the objects are initialized with the same set of values, while being instantiated by the `new` operator.

Note Instance method, `volComp()`, directly uses the instance variable: `length`, `breadth`, `height`, and `volume`. A very common mistake that many novice OOP programmers make, is to pass arguments to methods, multiply them and return the result. Although this might produce correct result but would not be correct OOPs approach as you are working with local variable rather than instance variables. Suppose if you create the `volComp()` method as shown below:

```
double volComp (double length, double breadth, double height){
      volume = length * breadth * height;
      return volume;
}
```

In this case, you are using local variables for calculating volume. The purpose is to calculate the volume of the room whose dimensions are already encapsulated in the Room object. So for that we need to access the instance variables as shown in Example 4.7 and not local variables. The usage of local variables defeats our purpose.

4.6.1 Parameterized Constructors

Just like methods, arguments can be passed to the constructors, in order to initialize the instance variables of an object. The above example had a limitation. Each `Room` has its own `length`, `breadth`, and `height` and it is very unlikely that each room is of the same size. In the previous example, all objects of `Room` class will have the same `volume` because the values for `length`, `breadth` and `height` are fixed for all objects. You can explicitly change them using an object instance, e.g.,

```
r1.length = 30
```

and then invoke the method `volComp` for calculating volume of the `Room`. But there should be a mechanism for specifying different values of instance variables for different objects of a class, as soon as the object is created. For example, if different dimensions can be specified for a `Room` then each `Room` will have its own volume. For this, the instance variables should be assigned a different set of values for different objects of the class. Hence we need to create a parameterized constructor that accepts arguments to initialize the instance variables with the arguments. Let us take an example to see how parameterized constructors can be used.

Example 4.8 Parameterized Constructor

```
L1    class Room2 {
L2    double length, breadth, height, volume;
L3    Room2(double l, double b, double h) {
L4       length = l;
L5       breadth = b;
L6       height = h;
L7    }
L8    // Computation of volume of the room
L9    double volComp(){
```

```
L10         volume = length * breadth * height;
L11         return volume;
L12     }
L13     public static void main (String args[ ]) {
L14         Room2 r1 = new Room2(14, 12, 10);
L15         Room2 r2 = new Room2(16, 15, 11);
L16         System.out.println("The volume of the room is " +r1.volComp());
L17         System.out.println("The volume of the room is " +r2.volComp());
L18     }
L19 }
```

Output

```
The volume of the room is 1680.0
The volume of the room is 2640.0
```

Explanation

Here we will explain only the relevant lines of the above example.

L3–7 The constructor `Room2` is defined, which has three arguments: `l`, `b`, and `h`, of type `double`. These are assigned to instance variables, `length`, `breadth` and `height`, respectively.

L13 Instance `r1` of class `Room2` is created. The values for the parameters are passed to the constructor in L3, with the invocation of the explicit constructor.

L14 Second instance `r2` of class `Room2` is created. Another set of values for the parameters is passed to the constructor in L3, with the invocation of the explicit constructor.

L15 & 16 The volumes for both the instances of `Room` are printed. You can see in the output that both the volumes are different, because different sets of parameters are used to calculate the volumes.

Note In the above program, we have created a parameterized constructor. If we create an object as shown below:

```
Room2 r3 = new Room2();
```

Instead of

```
Room2 r1 = new Room2(14, 12, 10);
```

The compiler will not compile this program. The obvious reason is that we have created a parameterized constructor in this class and we are trying to call the default constructor. Java states that if you provide a constructor for your class, the (automatically created) default constructor will not be provided to your class. And here we are invoking a no argument constructor which is neither explicitly created in our class nor will it be implicitly provided by Java.

4.6.2 Constructor Overloading

Just like methods, constructors can also be overloaded. Constructors are declared just as we declare methods, except that the constructors don't have any return type. Constructors for a class have the same name as that of the class, but they can have different signatures, i.e., different types of arguments or different number of arguments.

Such constructors can be termed as *overloaded constructors*. Constructors are differentiated on the basis of arguments passed to them.

In the example below, we have used two overloaded constructors, each having a different number of arguments, so that the JVM can differentiate between the various constructors.

Here we have two different classes, `Rectangle` and `ConstOverloading`. The `Rectangle` class has two constructors, both with the same name but different signatures. Each constructor is used for the initialization of instance variables.

Example 4.9 **Rectangle Class Depicting Constructor Overloading**

```
L1      Class Rectangle{
L2      int l, b;
L3      Rectangle(){
L4         l = 10;
L5         b = 20;
L6      }
L7      Rectangle(int x, int y){
L8         l = x;
L9         b = y;
L10     }
L11     int area()
L12     {
L13        return l * b;
L14     }
L15     }
```

Explanation

L3–5 Explicit default constructor is defined for the `Rectangle` class. This constructor initializes the instance variables with integer values.

L7–10 An overloaded constructor is defined for the `Rectangle` class, which accepts two integer values for initializing two instance variables.

L11–14 An instance method `area()` is defined to return the area of the rectangel.

Example 4.10 shows the second class `ConstOverloading`, which has the `main()`method inside it. While creating different instances of the `Rectangle` class, different overloaded constructors of the class are invoked with different number of parameters passed through the constructors. The values passed through the various constructors are used to initialize different instances of the `Rectangle` class.

Example 4.10 **Testing the Overloaded Constructors**

```
L1      class ConstOverloading {
L2      public static void main(String args[]) {
L3      Rectangle rectangle1 = new Rectangle();
L4  System.out.println("Area using first constructor:" +rectangle1.area());
L5  Rectangle rectangle2 = new Rectangle(4,5);
L6  System.out.println("Area using second constructor:" +rectangle2.area());
        }}
```

Output

```
Area using first constructor: 200
Area using second constructor: 20
```

Explanation

L3 The `Rectangle` object is created and the default constructor (i.e., no argument constructor, explicitly provided) is called.

L5 Another `Rectangle` object is created and the parameterized constructor is invoked. If there are a number of parameterized constructors in the class,

then which constructor will be invoked will depend upon the exact matching of the number of argument and the type of arguments in order. In our case, two integer arguments are passed, so a constructor is searched which accepts two integer arguments which is already defined in L7, Example 4.9.

The above example shows a case of overloaded constructors with differing number of arguments. Another case would be where different type of arguments can also be passed into the overloaded constructors.

4.7 CLEANING UP UNUSED OBJECTS

Many other object-oriented languages require that you keep a track of all the objects you create and that you destroy them when they are no longer needed. Objects are allocated memory from the heap memory and when they are not needed their allocated memory should be reclaimed. The clean-up code is tedious and often error-prone. Java allows programmer to create as many objects as they want, but frees them from worrying about destroying (deallocating memory) objects. The Java runtime environment deletes objects when it determines that they are no longer required. It has its own set of algorithms for deciding when the memory allocated to an object must be reclaimed. This automated process is known as *garbage collection*.

An object is eligible for garbage collection when no references exist on that object. References can be either implicitly dropped when it goes out of scope or explicitly dropped by assigning `null` to an object reference.

4.7.1 Garbage Collector

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer needed. Two basic approaches used by garbage collectors are *Reference counting* and *tracing*. Reference counting maintains a reference count for every object. A newly created object will have count as 1. Throughout its lifetime, the object will be referred to by many other object thus incrementing the reference count and as the referencing object move to other objects, the reference count for that particular object is decremented. When reference count for a particular object is 0, the object can be garbage collected.

Tracing technique traces the entire set of objects (starting from root) and all objects having reference on them are marked in some way. Tracing garbage collector algorithm popularly known as is *mark and sweep* garbage collector scans Java's dynamic memory areas for objects, marking those objects that are referenced. After all the objects are investigated, the objects that are not marked (not referenced) are assumed to be garbage and their memory is reclaimed. Mark and sweep collectors further use the techniques of `Compaction` and `Copying` for fragmentation problems (refer to memory management in operating system for details) that may arise once you sweep the unreferenced objects. Compaction moves all the live objects towards one end making the other end a large free space and copying techniques copies all live objects besides each other into a new space and the old space is considered free now.

The garbage collector runs either synchronously or asynchronously in a low priority daemon thread. The garbage collector executes *synchronously* when the system runs out of memory or *asynchronously* when the system is idle. The garbage collector can be invoked to run at any time by calling `System.gc()` or `Runtime.gc()`. But asking the garbage collector to run does not guarantee that your objects will be garbage collected.

4.7.2 Finalization

Before an object gets garbage collected, the garbage collector gives the object an opportunity to clean up itself through a call to the object's `finalize()` method. This process is known as *finalization*.

All occupied resources (sockets, files, etc.) can be freed in this method. The `finalize()` method is a member function of the predefined `java.lang.Object` class. A class must override the `finalize()`method to perform any clean up if required by the object.

4.7.3 Advantages and Disadvantages

There are many advantages of using garbage collection apart from freeing the programmer from worrying about deallocation of memory. It also helps in ensuring integrity of programs. There is no way by which Java programmers can knowingly or unknowingly free memory incorrectly.

The disadvantage of garbage collection is the overhead to keep track of which objects are being referenced by the executing program and which are not being referenced. The overhead is also incurred on finalization and freeing memory of the unreferenced objects. These activities will incur more CPU time than would have been incurred if the programmers would have explicitly deallocated memory.

4.8 CLASS VARIABLES AND METHODS—`static` KEYWORD

When we create an object, a primitive type variable, or call a method, some amount of memory is set aside for the said object, variable, or method. Different objects, variables, and methods will occupy different areas of memory when created/called. Sometimes we would like to have multiple objects, shared variables, or methods. The `static` keyword effectively does this for us. It is possible to have *static methods* and *variables*.

Before going further, we must discuss the kind of variables Java supports. These include: *local variables*, *instance variables*, and *class/static variables*.

Local Variables Local variables are declared inside a method, constructor, or a block of code. When a method is entered, its contents (values of local variables) are pushed onto the call stack. When the method exits, its contents are popped off the stack and the memory in stack is now available for the next method. Parameters passed to the method are also local variables which are initialized from the actual parameters. The scope of local variables is limited to the method in which they have been defined. They have to be declared and initialized before they are used. Access specifiers like `private`, `public`, and `protected` cannot be used with local variables.

Instance Variables Instance variables are declared inside a class, but outside a method. They are also called *data member*, *field*, or *attributes*. An object is allotted memory for all its instance variables on the heap memory. As objects instance variables remain live as long as the object is active. They are accessible directly in all the instance methods and constructors of the class in which they have been defined. By default, they are initialized to their default values according to their respective types.

Class/static Variables Class/static variables declaration is preceded with the keyword `static`. They are also declared inside a class, but outside a method. The most important point about static variables is that there exists only a single copy of static variables per class. All objects of the class share this variable. Static variables are normally used for constants. By default, static variables are initialized to their default values according to their respective types.

Note No variable can have an undefined value. Instance or class variables are implicitly initialized to their respective default values, whereas local variables are not implicitly initialized to a default value and must be explicitly initialized in Java.

4.8.1 Static Variables

Java does not allow global variables. The closest thing we can get to a global variable in Java is to make the instance variable in the class `static`. The effect of doing this is that when we create multiple objects of that class, every object shares the static variable, i.e. there is only one copy of the variable declared as `static`. To make an instance variable static we simply precede the declaration with the keyword `static`.

```
static int var = 0;
```

In effect, what we are really doing is that this instance variable, `var`, no matter how many objects are created, should always reside in the same memory location, regardless of the object. This then simulates like a 'global variable.' We usually declare a variable as final and static as well, since it makes sense to have only one instance of a constant. It is worthwhile to note that people refer to static instance variables as 'class variables.' Before proceeding further, let us take an example to depict how static variables are declared.

Note **Instance Variables vs Class Variables**

Class variables can be declared with the '`static`' keyword. For example,

```
static int y = 0;
```

All instances of the class share the static variables of the class. A class variable can be accessed directly with the class name, without the need to create an instance.

Without the 'static' keyword, it is called an 'instance variable' and each instance of the class has its own copy of the variable.

Example 4.11 **Instance and Class Variables**

In the following code, the class `Test1` has two variables, `x` and `y`.

```
L1   class Test1 {
L2   int x = 0;          // instance variable
L3   static int y = 0;   // class variable

     //setter methods
L4   void setX (int n) {x = n;}
L5   void setY (int n) {y = n;}

     //getter methods
L6   int getX() { return x;}
L7   int getY() { return y;}
     }
```

We could have another class `Test2` having the `main()` function where the use of static variable declared in the class `Test1` can be shown:

Example 4.12 **A Class Showing the Use of Class (Static) Variables**

```
L1  class Test2 {
L2  public static void main(String[] arg){
L3     Test1 t1 = new Test1();
L4     Test1 t2 = new Test1();
L5     t1.setX(9);
L6     t2.setX(10); // object t1 and t2 have separate copies of x
L7  System.out.println("Instance variable of object t1 : " +t1.getX());
L8  System.out.println("Instance variable of object t2 : " +t2.getX());
    // class variable can be accessed directly through Class Name
    // (if changed to Test2.x, it won't compile)
L9     System.out.println("Value of y accessed through class Name: " +Test1.y);
L10    Test1.y = 7;
L11    System.out.println("Changed value of y accessed through class Name: " +Test1.y);
    // class variable can be manipulated thru methods as usual
L12    t1.setY(Test1.y+1);
    // class variable can be accessed through objects also
L13    System.out.println("Value of y accessed through object t2:" +t2.getY());
L14    }
L15 }
```

Output

```
Instance variable of object t1 : 9
Instance variable of object t2 : 10
Value of y accessed through class Name: 0
Changed value of y accessed through class Name: 7
Value of y accessed through object t2: 8
```

Explanation

L7 Output printed is 9, i.e., the instance variable is printed with the help of the object.

L8 Output printed is 10, i.e., another instance variable is printed with the help of the object.

L9 Output printed is 0. It is important to note that here, we need not have an object for class `Test 1` to access the static variable of `Test 1` (refer to L3 of class `Test 1`).

L10 Static variable is assigned a value 7 using the class name itself.

L11 Output printed is 7.

L12 Instance method `setY()` is invoked using the object `t1`, where the value of static variable, `y` (i.e., 7), accessed through class name `Test1` is incremented by 1 and passed as argument.

L13 Output printed is 8, as `t2.getY()`returns the value set by `t1.setY()`in L12. This is done to show that the value of `y` is being shared by all the objects of the class, as it is a static variable.

4.8.2 Static Methods

Like static variables, we do not need to create an object to call our static method. Simply using the class name will suffice. Static methods however can only access static variables directly. Variables that have not been declared static cannot be accessed by the static method directly, i.e., the reason why we create an object of the class within the main (which is static) method to access instance variables and call instance methods. To make a method static, we simply precede the method declaration with the keyword `static`.

```
static void a Method(int param1) { ..... ..... }
```

Note **Instance Method vs Class Method**

`static` methods can be accessed through the class name itself. Methods declared without the `static` keyword (instance methods) can be accessed using the object/instance of the residing class. `static` methods are also known as class methods.

`static` methods can call other static methods directly. If a static method to be invoked is within the same class, then only the `static` method name can be mentioned to invoke it. Else if the `static` method is outside the class, then the class name has to be prefixed with the `static` method name to invoke it. But invoking `non-static` methods (instance methods) from static methods requires an instance of the class. Also note that methods declared as `static` cannot access the variables declared without the `static` keyword. It is quite evident in the following example where it gives a compilation error, un less `x` is also static.

```
class Test {
int x = 3;
static int returnX(){
    return x;
}
public static void main(String args[])
{
System.out.println(returnX()); // static method invoked directly
}}
```

Let us take an example to show the use of static methods.

Example 4.13 **A Class Having Static Members**

```
L1      class Area {
L2      static int area; // class variable
L3      static int computeArea (int width, int height){
L4          area = width * height;
L5          return area;
L6          }
L7      }
```

The above class `Area` has a class variable declared in L2 and a static method, `computeArea()` with two arguments in L3.

Example 4.14 **Calling Static Method from Another Class**

```
L1    class CallArea{
L2    public static void main(String args[]){
L3        System.out.println(Area.computeArea(4,3));
L4        }
L5    }
```

Output

```
12
```

Explanation

L3 The method `computeArea()` of class `Area` is being called without referencing it through any object/ instance. Instead, it can be invoked using that class name only, which it belongs to. It is so because this method has been declared as static in L3 of class `Area`. The return value of the method is printed using `System.out.println()`.

4.8.3 Static Initialization Block

A block of statements can be enclosed in parenthesis with `static` keyword applied to it. This block of statement is used for initializing static or class variables. If the initialization logic is simple, the class variables can be assigned values directly but in case some logic is used for assigning values to the variables, static blocks can be used. The syntax for static block is as follows:

```
static
{
  ...
}
```

The `static` executes as soon as the class loads even before the JVM executes the main method. There can be any number of `static` blocks within the class and they will be executed in the order in which they have appeared in the source code.

Note In case the `static` keyword is dropped from this block, it becomes an instance initialization block and all code placed inside this block is placed inside the constructors before the source code written in the constructor by the Java compiler. Actually the code of instance initialization block is placed in the <init> method, which is created for every constructor by the compiler, before the source code mentioned by programmer in the constructor.

Let us take an example to see how `static` block, instance initialization block, instance variable initializes and constructor executes. The program clearly shows that static block executes even before main method. This program also includes an instance variable instance initialization blocks with a constructor. Both the instance block and the constructor code gets invoked as soon as the object of the class is created. How? As already stated, the code of initializer instance initialization block is placed within the constructor, before the constructors own code, by Java compiler. This is evident by seeing the output, the print statement in the instance initialization block executes before statement mentioned in the constructor. The static block also shows declaration of a variable which is local to the block.

Example 4.15 **Static Initialization Block, Instance Initialization Block and Constructor**

```
class StaticBlockDemo
{
```

```
    int x=10;   // instance variable initializer
    /* static initialization block */
    static
    {

         int z=10; // local variable
         System.out.println("In static block");
    }
    // Instance initialization block
    {
         System.out.println("In Instance Initialization block");
         System.out.println("Printing Instance variable Initializer
         value through Block: " +x);
    }
    // Constructor
    StaticBlockDemo(int y)
    {
         System.out.println("Within Constructor");
         System.out.println("Instance variable printed using constructor: "+x);
         x=y;
         System.out.println("Instance variable initialized using constructor: "+x);
    }
         /* To see whether the code of instance variable initializer
         and block is copied within every constructor we create another
         constructor and see the output. The following constructor when
         invoked also prints the contents of instance variable intializer
         and block.  So the contents of instance variable initializer and
         block are copied in every constructor by the compiler. In other
         word, they are copied in every <init> method created for every
         constructor before the constructors own code.*/

    StaticBlockDemo()
    {
         System.out.println("Within Constructor");
         System.out.println("Instance variable printed using constructor: "+x);
    }
    public static void main(String[] args)
    {
            System.out.println("In main");
            StaticBlockDemo st = new StaticBlockDemo(100);
            System.out.println("----------------------------------------");
            StaticBlockDemo st1 = new StaticBlockDemo();
    }
}
```

Output

```
D:\javaprg>java StaticBlockDemo
In static block
In main
In Instance Initialization block
Printing Instance variable Initializer value through Block: 10
Within Constructor
Instance variable printed using constructor: 10
Instance variable initialized using constructor: 100
----------------------------------------
In Instance Initialization block
Printing Instance variable Initializer value through Block: 10
Within Constructor
Instance variable printed using constructor: 10
```

4.9 `this` KEYWORD

The keyword `this` is used in an instance method to refer to the object that contains the method, i.e., it refers to the current object. Whenever and wherever a reference to an object of the current class type is required, `this` can be used.

It can also differentiate between instance variables and local variables. Let us revisit the code segment of Example 4.8. Here the use of `this` will make you understand its use.

```
L3    Room2(double l, double b, double h){
L4    this.length = l;
L5    this.breadth = b;
L6    this.height = h;
L7    }
```

Here, the use of `this` does not do anything differently than the earlier code in Example 4.8. It is perfectly legitimate to use it in the way it has been done. Inside `Room2`, `this` will always refer to the current object, of `Room2`. The obvious question that would arise is when and why should we use `this` in an application?

The exact purpose of `this` is to remove ambiguity between local and instance variables. In Example 4.8, we had three instance variables declared in L2. Look carefully. The formal (local variables) parameters of `Room()` in L3 have different names, (`l`, `b`, and `h`) from the instance variables (`length`, `breadth`, and `height`).The values of these formal parameters are passed to the instance variables. If a like names are provided for both the parameters (formal and instance variables) then the instance variables will be hidden (or shadowed) by the local variables. Suppose the formal parameters had been named as `length`, `breadth`, and `height`, which are also the names of the instance variables used in the class, then it is difficult to distinguish between local variable and instance variable as shown below:

```
Room2 (double length, double breadth, double height){
      length = length;
      breadth = breadth;
      height = height;
      }
```

It is an ambiguous situation for the JVM as it does not understand what has to be done; whether instance variables have to be initialized with formal parameters or vice versa. The problem arises because JVM cannot clearly distinguish which is a local variable and which is an instance variable. In this case the local variables shadow or hide the instance variables. If you try to access or print the `length` variables in the constructor `Room2`, the local variable `length` will be printed and not the instance variable: `length`, this allows you to solve the problem of a variable's scope, because it lets you refer to the object directly. `this` keyword makes a clear cut distinction between local and instance variable. `this.length` refers to the `length` instance variable of the current object. The above block of code can be re-written as follows.

```
Room2 (double length, double breadth, double height){
     this.length = length;
     this.breadth = breadth;
     this.height = height;
     }
```

In the above code it is clearly evident local variable `length` value should be assigned to the instance variable `length` of the current object and soon for other variables.

Hence, the names of instance variables and the formal parameters can be kept similar because `this` has made it possible for the JVM to differentiate between instance and local variables. Still, one can argue that a programmer can very well use different variable names for instance and local variables.

Constructor Chaining It means a constructor can be called from another constructor. Let us revisit Example 4.8.

```
/* First Constructor */
     Room2( )
     {

            // constructor chained
            this(14,12,10);
     }
/* Second Constructor */
     Room2 (double l, double b, double h)
     {
     length = l;
            breadth = b;
            height = h;
     }
```

In the above code, two constructors have been created: one without arguments and another with three arguments. In the first constructor, we have used `this` keyword to call the second constructor and passed the required arguments in the call to second constructor.

Whenever we create an object of the class `Room2` as,

```
Room2 r1 = new Room2();
```

the first constructor will be invoked which is chained to the second constructor.

4.10 ARRAYS

Till now, we have discussed how to declare variables of a particular data type, which can store a single value of that data type. The allocation of memory space, when a variable is declared, cannot further be sub-divided to store more than one value. There are situations where we might wish to store a group of similar type of values in a variable. It can be achieved by a special kind of data structure known as *arrays*.

An *array* is a memory space allocated that can store multiple values of same data type in contiguous locations. This memory space, which can be perceived to have many logical contiguous locations, can be accessed with a common name. For example, we can define an array as 'marks' to represent a set of marks of a group of students. Now the next question is how to access a particular value from a particular location? A specific element in an array is accessed by the use of a subscript or an index used inside the brackets, along with the name of the array. For example, `marks[5]` would store the marks of the fifth student. While the complete set of values is called an array, the individual values are known as *elements*. Arrays can be two types:

- one dimensional array
- multi-dimensional array

4.10.1 One-dimensional Arrays

In a one-dimensional array, a single subscript or index is used, where each index value refers to an individual array element. The indexation will start from 0 and will go up to $n-1$, i.e., the first value of the array will have an index of 0 and the last value will have an index of $n-1$, where n is the number of elements in the array. So, if an array named marks has been declared to store the marks of five students, the computer reserves five contiguous locations in the memory, as shown in Fig. 4.4.

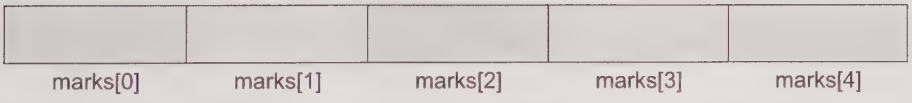


Fig. 4.4 Marks Array

Suppose, the five marks to be assigned to each array element are 60, 58, 50, 78, and 89. It will be done as follows:

```
Marks[0] = 60;
Marks[1] = 58;
Marks[2] = 50;
Marks[3] = 78;
Marks[4] = 89;
```

Figure 4.5 shows the marks array with data elements.

Fig. 4.5 Marks Array Having Data Elements

Creation of Array

Creating an array, similar to an object creation, can inherently involve three steps:

- Declaring an array
- Creating memory locations
- Initializing/assigning values to an array

Declaring an Array Declaring an array is same as declaring a normal variable except that you must use a set of square brackets with the variable type. There can be two ways in which an array can be declared.

- `type arrayname[];`
- `type[] arrayname;`

So the above `marks` array having elements of integer type can be declared either as

```
int marks[];
```

or

```
int[] marks;
```

Creating Memory Locations An array is more complex than a normal variable, so we have to assign memory to the array when we declare it. You assign memory to an array by specifying its size. Interestingly, our same old `new` operator helps in doing the job, just as shown below:

```
Arrayname = new type [size];
```

So, allocating space and size for the array named as `marks` can be done as,

```
marks = new int[5];
```

Both (declaration of array and creation of memory location), help in the creation of an array. These can be combined as one statement, for example,

```
type arrayname[] = new type[];
```

or

```
type[] arrayname = new type[];
```

It is interesting to know what the JVM actually does while executing the above syntax. During the declaration phase, `int marks[];`

Figure 4.6 shows the marks array after memory is allocated to the array on execution of the following statement:

```
marks = new int[5];
```

Here is an example to show how to create an array that has 5 marks of integer type.

```
class Array{
public static void main(String[]
args){
    int[] marks = new int[5];
     }
}
```

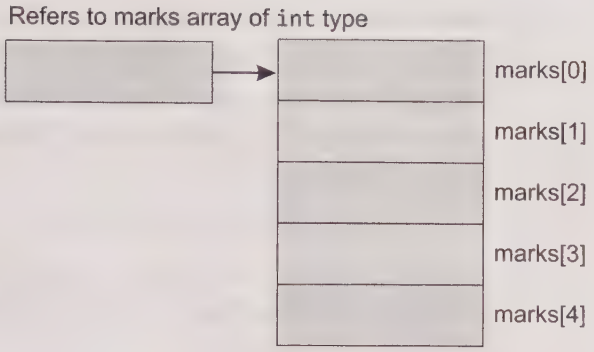


Fig. 4.6 Creation of Arrays

Initializing/assigning Values to an Array Assignment of values to an array, which can also be termed as initialization of array, can be done as follows:

```
Arrayname[index] = value;
```

We have just discussed how to create a list of parameters to be assigned in an array. Example 4.16 shows how to set the values for an array of 5 marks (Fig. 4.6).

Example 4.16 Setting Values in an Array

```
L1  class Array {
L2  public static void main(String[] args){
L3    int[] marks = new int[5];
L4    marks[0] = 60;
L5    marks[1] = 58;
L6    marks[2] = 50;
L7    marks[3] = 78;
L8    marks[4] = 89;
L9    }
L10 }
```

Arrays can alternately be assigned values or initialized in the same way as the variables, i.e., at the time of declaration itself. The syntax for the same can be,

```
type arrayname[] = {list of values};
```

For example, `int marks[] = {60, 58, 50, 78, 89}`

Here, the `marks` array is initialized at the time of creation of array itself. The above statement does the same thing as the code between L3 to 8 of Example 4.16. An example of array creation and initialization is given below.

Example 4.17 Creation and Initialization of an Array

```
class Array
{
public static void main(String[] args){
  int[] marks = {60, 58, 50, 78, 89};
}
}
```

How to Use for Loops with Arrays?

The `for` loops can be used to assign as well as access values from an array. To obtain the number of values in an array, i.e., *the length of the array,* we use the name of the array followed by the `dot` operator and the variable `length`. This `length` property is associated with all the arrays in Java. For example,

```
System.out.println("There are " + marks.length + "in the array");
```

will print the number of elements in the `marks` array, i.e., 5. Example 4.18 shows how to use a `for` loop to set all the values of an array to 0 which, you will see, is much easier than setting all the values to 0 separately.

Example 4.18 Setting Values in an Array Using for Loop

```
L1    class Array {
L2    public static void main(String[] args){
L3      int[] marks = new int[5];
L4      for (int i = 0; i<marks.length; i++)
L5      marks[i] = 0;
L6      }
L7    }
```

Explanation

L3 Creates an array `marks`, having five locations to store five elements.

L4 `i` signifies the subscript of the array, which is always an integer type. The `for` loop starts with the first location of the array, it stands at the 0th subscript and iterates by 1 up to the last location of the array, which is returned by `marks.length`.

Various operations can also be performed on the values of an array, which can again be assigned to the array. For example, the following code increments all the marks in the class by 5.

Example 4.19 Incrementing the Values of Data Elements in an Array

```
for (inti = 0; i<grades.length; i++){
   grades[i] = marks[i] + 5;
}
```

To access a particular value in the array, we use the name of the array, followed by an open bracket, followed by an expression that gives the index, followed by a close bracket. For example, here is a simple code to print all the marks in the array of marks declared above.

Example 4.20 Printing the Values of Data Elements of an Array

```
for (inti = 0; i<marks.length; i++){
   System.out.println(marks[i]);
}
```

Sorting an Array Let us take an example where we apply all the concepts of array that we have learnt until now. If we have been given a set of marks and we have to sort the marks in ascending order.

Example 4.21 Sorting an Array

```
class SortArray{
public static void main(String[] args)
{
   int[] marks = {3, 5, 1, 2, 4};
   int temp, n;
   n = marks.length;
   System.out.print("The list of marks is: ");
   for(int i =  0; i< n; i++){
   System.out.print(marks[i]+ " ");
}

for (int i = 0; i< n; i++){
for (int j = i+1; j < n; j++){
if (marks[i] < marks[j])
{
   temp = marks[i];
   marks[i] = marks[j];
   marks[j] = temp;
}
}
}
System.out.print("\nList of marks sorted in descending order is: ");
for (int i = 0; i< n; i++)
{
   System.out.print(marks[i]+" ");
}
}
}
```

Output

```
c:\javabook\programs\chap4>java SortArray
The list of marks is: 3 5 1 2 4
List of marks sorted in descending order is: 5 4 3 2 1
```

Explanation

L1 Class `SortArray` declared.

L2 `main()`declared and its body starts with left {.

L3 Array named `marks` created with initialized values.

L4 Instance variables, `temp` and `n` declared to be integer type.

L5 Length of the array is stored in `n`.

L7–9 `for` loop is used to print the values of the original list, i.e., `marks`.

L10 Defines `for` loop which iterates from 0 to length of the array –1.

L11 A nested `for` loop is declared which iterates from i + 1 to n –1. L12–16 are part of the inner `for` loop, and these statements are executed for each

value of i from 0 to n –1 and j from 1 to n–1 as shown in Fig. 4.7 below.

L12–16 In the first iteration, value of i is 0 and j is 1. The marks at the 0th index are compared with the marks at the first index. If marks at 0th index are less than marks at the 1st index they are swapped. For swapping, a temporary variable named temp is created (L13). Marks at ith index (first iteration value of i is 0) are assigned to temp (L14). The marks at jth (first iteration value of j is 1) index are assigned to marks at marks at ith index (L15) and marks in temporary variable are assigned to marks at the jth position. Thus the value of jth position is swapped with the value at ith position. Figure 4.7 illustrates how the outer and inner loops execute for each value of i and j. It also shows when the values of the array are swapped.

L20–22 Display the sorted array. The array has been sorted in descending order.

When i = 0		
j = 1	marks[0] < marks[1] (3) (5)	Yes, so they are swapped New Array is 5, 3, 1, 2, 4
j = 2	marks[0] < marks[2] (5) (1)	No, not swapped New Array is 5, 3, 1, 2, 4
j = 3	marks[0] < marks[3] (5) (2)	No, not swapped New Array is 5, 3, 1, 2, 4
j = 4	marks[0] < marks[4] (5) (4)	No, not swapped New Array is 5, 3, 1, 2, 4
When i = 1		
j = 2	marks[1] < marks[2] (3) (1)	No, not swapped New Array is 5, 3, 1, 2, 4
j = 3	marks[1] < marks[3] (3) (2)	No, not swapped New Array is 5, 3, 1, 2, 4
j = 4	marks[1] < marks[4] (3) < (4)	Yes, swapped New Array is 5, 4, 1, 2, 3
When i = 2		
j = 3	marks[2] < marks[3] (1) < (2)	Yes, swapped New Array is 5, 4, 2, 1, 3
j = 4	marks[2] < marks[4] (2) < (3)	Yes, swapped New Array is 5, 4, 3, 1, 2
When i = 3		
j = 4	marks[3] < marks[4] (1) < (2)	Yes, swapped New Array is 5, 4, 3, 2, 1
When i = 4		
j = 5	Inner for loop does not execute.	
When i = 5, Outer for loop exits		

Fig. 4.7 Execution of Loops in SortArray Example

4.10.2 Two-dimensional Arrays

Sometimes values can be conceptualized in the form of a table that is in the form of rows and columns. Suppose we want to store the marks of different subjects. We can store it in a one-dimensional array.

Now if we want to add a second dimension in the form of roll no of the student. This is possible only if we follow a tabular approach of storing data, as shown in Table 4.4.

You can easily notice that Table 4.3 can store only subject names and the marks obtained by one student, while Table 4.4 can store the details of multiple students. There can be enumerable such situations where we can use a two-dimensional structure. Java provides a solution for the storage of such a structure in the form of two-dimensional arrays.

If you want a multidimensional array, the additional index has to be specified using another set of square brackets. The following statements create a two-dimensional array, named as `marks`, which would have 4 rows and 5 columns, as shown in Table 4.4.

Table 4.3 One-dimensional Marks Array

Subjects	Marks
Physics	60
Chemistry	58
Mathematics	50
English	78
Biology	89

Table 4.4 Two-dimensional Marks Array

Subject / Roll No.	Physics	Chemistry	Mathematics	English	Biology
01	60	67	47	74	78
02	54	47	67	70	67
03	74	87	76	69	88
04	39	45	56	55	67

```
int marks[][]          //declaration of a two-dimensional array
marks = new int[4][5]; //reference to the array allocated, stored in marks
                         variable
```

This is done in the same way as it has already been explained while discussing one-dimensional arrays. The two statements, used for array creation, can be merged into one as,

```
int marks[][] = new int[4][5];
```

Another way of representing the above statement can be,

```
int[][] marks = new[4][5];
```

This statement just allocates a 4 × 5 array and assigns the reference to the array variable `marks`. The first subscript inside the square bracket signifies the number of rows in the table or matrix and the second subscript stands for the number of columns. This 4 × 5 table can store 20 values altogether. Its values might be stored in contiguous locations in the memory, but logically, the stored values would be treated as if they are stored in a 4 × 5 matrix. Table 4.5 shows how the `marks` array is conceptually placed in the memory by the above statement.

Table 4.5 4 × 5 Marks Array

60 (marks[0][0])	67 (marks[0][1])	47 (marks[0][2])	74 (marks[0][3])	77 (marks[0][4])
54 (marks[1][0])	47 (marks[1][1])	67 (marks[1][2])	70 (marks[1][3])	67 (marks[1][4])
74 (marks[2][0])	87 (marks[2][1])	76 (marks[2][2])	69 (marks[2][3])	88 (marks[2][4])
39 (marks[3][0])	45 (marks[3][1])	56 (marks[3][2])	55 (marks[3][3])	67 (marks[3][4])

Like a one-dimensional array, two-dimensional arrays may be initialized with values at the time of their creation. For example,

```
int marks[2][4] = {2, 3, 6, 0, 9, 3, 3, 2};
```

This declaration shows that the first two rows of a 2 × 4 matrix have been initialized by the values shown in the list above. It can also be written as,

```
int marks[][] = {(2, 3, 6, 0), (9, 3, 3, 2)};
```

In the above declaration, subscripts need not be shown, as it is evident from the manner in which the list of values have been presented. Here, the list of values has two different sets of values, separated by a comma, each standing for a row.

It is important to understand how Java treats 2-D arrays. 2-D arrays are treated as 1-D array. For example, the above declaration of 2 × 4 array will create three 1-D array. One for storing the number of row arrays (i.e. 2) and the other two arrays will be used for storing the contents of the rows. The size of these two arrays will be 4. As shown in Fig. 4.7, the size of row array is the number of rows and each field in the row array points to a 1-D array that contains the column values for the rows. So `marks[0][0]`will have the value 2, `marks [0][1]` will have 3, `marks[1][0]` with 9, and so on.

Assigning and accessing the values in a two-dimensional array is done in the same way, as was done in a one-dimensional array. The only difference is that, here you have to take care of the positional values of the array using two subscripts (shown in square brackets), while in a one-dimensional array, only one subscript was used for the purpose. Table 4.5 shows the positional values of a two-dimensional array.

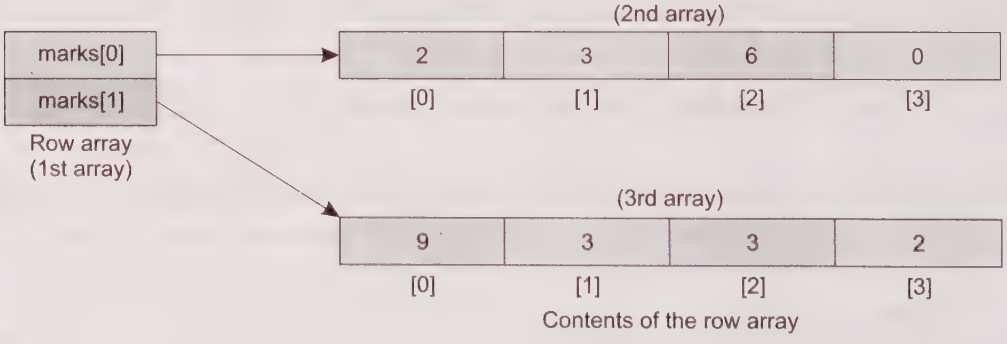


Fig. 4.8 2-D Array

All that you need to do to create and use a 2-D array is to use two square brackets instead of one.

Example 4.22 **Setting Values in a Two-dimensional Array**

```
L1  class DemoArray1 {
L2  public static void main(String[] args) {
L3  int a[][]  = new int [2][];
L4  /* int a1[][]  = new int [][2]; */
L5  int m[][] = {{2,3,6,0},{9, 3, 3, 2}};
L6  for(int i=0;i<m.length;i++)
      {
L7        for(int j=0;j<m[i].length;j++)
L8        System.out.print(m[i][j] +" ");
L9    System.out.println();
      }
    }}
```

Output

```
C:\javabook\programs\chap4>java DemoArray1
2 3 6 0
9 3 3 2
```

Explanation

L3–4 Shows the declaration of a 2-D integer array having two rows. The number of columns is not specified but the reverse declaration is illegal as shown in L4.

L5 Shows a declaration of a 2-D array with values assigned to it. No number has been specified in the row and column square brackets of array `m`. The rows and columns are decided on the basis of how the values are passed to the array. `m` is having 2 rows and 4 columns. The number of inner curly bracket (opening and closing) determines the number of rows (row array) and the number of individual values in a particular curly bracket (opening and closing) will determine the columns in a row (separate 1D array will be created for each row). Each index in a row array will point to a column array.

L6 A `for` loop is created. This `for` loop is used for iterating through the row array and that is why it iterates from 0 to the length of the array `m`.

L7 An inner `for` loop is created for iterating the columns in a row array. The inner `for` loop iterates from 0 to the length of the 1D array pointed by the individual fields in the row array. That is why the loop iterates up to `m[i].length`.

L8 Prints the individual items of the array at all row and column combinations.

Let us take a more complex but useful example of matrix multiplication. Two matrices are to be multiplied, so two arrays capable of holding the same number of rows and columns as matrices are required.

Example 4.23 Matrix Multiplication

```
L1  class MatrixMul {
L2  public static void main(String args[]) {
L3     int array[][] = {{3,7},{6,9}};
L4     int array1[][] = {{5,4},{3,6}};
L5     int array2[][] = new int[2][2];
L6     int x = array.length;
L7     System.out.println("Matrix 1: ");
L8     for (int i=0; i<array.length; i++) {
L9     for (int j=0; j<array[i].length; j++) {
L10         System.out.print(" "+array[i][j]);
L11    }
L12    System.out.println();
L13    }
L14    int y = array1.length;
L15    System.out.println("Matrix 2: ");
L16    for (int i=0; i<array1.length; i++) {
L17    for (int j=0; j<array1[i].length; j++)
       {
L18         System.out.print(" "+array1[i][j]);
L19    }
L20    System.out.println();
L21    }
L22    for (int i=0; i<x; i++) {
L23    for (int j=0; j<y; j++) {
L24    for(int k=0; k<y; k++) {
L25                array2[i][j] += array[i][k]*array1[k][j];
L26                }
L27         }
L28    }
L29    System.out.println("Multiplication of both matrices: ");
L30    for (int i=0; i<x; i++) {
L31    for (int j=0; j<y; j++) {
L32                          System.out.print(" "+array2[i][j]);
L33                }
L34                System.out.println();
L35         }
L36    }
L37 }
```

Output

```
Matrix 1:
 3 7
 6 9
Matrix 2:
 5 4
 3 6
Multiplication of both matrices:
 36 54
 57 78
```

Like 2-D arrays, we can define any multidimensional array having *n* dimensions. While declaring an *n*-dimensional array, *n* number of square brackets will be used. All the operations in any type of multidimensional array will be similar to that of a one-dimensional or two-dimensional array.

4.10.3 Using for-each with Arrays

The enhanced `for` loop, i.e., `for-each` was introduced in Java 5 to provide a simpler way to iterate through all the elements of an array or a collection. The format of `for-each` is as follows:

```
for (type var : arr){
    // Body of loop
}
```

For example, we can use `for-each` loop to calculate the sum of elements of an array as follows:

```
int[] arr = {2,3,4,5,6};
int sum = 0;
for(int a : arr) // a gets successively each value in arr
{
     sum += a;
}
```

The disadvantage of `for-each` approach is that it is possible to iterate in forward direction only by single steps.

4.10.4 Passing Arrays to Methods

Arrays can be passed to methods as well. The following example shows a two-dimensional array being passed to a method. The static method displays the contents of that array.

Example 4.24 Arrays as Arguments to Methods

```
L1  class PassingArray
L2  {
L3     static void show(int[][] a)
L4     {
L5        for(int i=0;i<a.length;i++)
L6        {
L7           for(int j=0;j<2;j++)
L8           {
L9                System.out.print(" " +a[i][j]);
L10          }
L11       System.out.println();
L12       }
L13    }
L14    public static void main(String args[])
L15    {
L16       int a[][]={{1,2},{2,3}};
L17       show(a);
L18    }
L19 }
```

Output

```
D:\javabook\programs\chap4\PassingArray
1 2
2 3
```

Explanation

L1 Class declaration.
L3 Declares a static method 'show' that accepts an argument i.e., a two-dimensional array.
L5 Shows a for loop that would loop from 0 to the length of array. This for loop is basically used to refer to the first dimension of the 2D array.
L6 Marks the beginning of for loop defined in L5.
L7 Shows another for loop that would represent the second dimension. Our 2D array has only two elements as each array item. So the index for referencing any individual element would be a[0][0] or a[0][1] for the first row of the Array. Subsequently The next row items can be referenced as a [1][0] and a [1][1] and so on (a [2][0], a [2][1] etc.). As is evident, the second index does not go beyond 1, so we have declared a for loop, in this statement, that iterates for less than 2 times.
L9 Prints the individual elements of the array based on the values of indexes set by the values of i and j.
L10 Marks the closure of the inner for loop.
L11 Is a simple print statement used for formatting the output. This will move the cursor to new line. Basically it is used to show the individual elements of the array on a new line.
L12 Marks the closure of the outer for loop.
L13 Ends the method show.
L14 Main method declaration.
L16 An int array is defined and initialized with values.
L17 Static method show is invoked and array is passed as an argument to it. As show is a static method it can be invoked directly.
L18 & 19 Ends the main method and the class.

4.10.5 Returning Arrays from Methods

Arrays can not only be passed to methods but we can use arrays as return value from methods. If you are faced with a situation where you want to return multiple values from a method, all the values can be encapsulated in an array and returned. The following example shows a two dimensional array being returned from a method. The main method displays the contents of that array.

Example 4.25 Returing Multiple Values

```
class ReturningAnArray
{
   // static method declared to return a 2D Array
   static int[][] show(){
   int a[][]={{1,2},{2,3}};
   return a;
}
public static void main(String args[])
{
   int a[][]=show();    // return value is captured in a 2D Array
   for(int i=0;i<a.length;i++)
   {
        for(int j=0;j<2;j++)
        {
               System.out.print(" " +a[i][j]);
```

```
            }
            System.out.println();
            }
        }
    }
```

Output

```
D:\javabook\programs\chap4\ReturningAnArray
1 2
2 3
```

Explanation

This program is almost the same as that of previous program. The difference is that the `show` method now returns a two-dimensional array. The logical steps to iterate the two-dimensional array is same as that of previous program with a change i.e., now they belong to `main` method instead of `show` method.

4.10.6 Variable Arguments

Variable arguments can be used when the number of arguments that you want to pass to a method are not fixed. The method can accept different number of arguments of same type whenever they are called. The generic syntax for this notation is:

```
returntype methodName(datatype...arrayname)
```

Let us take an example to show how it can be implemented.

Example 4.26 Variable Arguments

```
class VarArgs
{
   // integer variable argument used in add method
   static void add(int...a)
   {
      int sum=0;
      for(int i=0;i<a.length;i++)
      sum=sum+a[i];
      System.out.println("SUM = "+sum);
   }

   // variable arguments syntax used in main method
   public static void main(String...args)
   {

   // four arguments are passed to the add method
      add(2,3,4,5);
      // Three arguments are passed to the same add method
      add(2,3,4);
   }
}
```

Output

```
D:\javabook\programs\chap 4\java VarArgs
SUM = 14
SUM = 9
```

4.11 COMMAND-LINE ARGUMENTS

You must be aware of the basic DOS commands. Have you ever used the command to move a file from one location to another, say `abc.txt` from C:\ to D:\, `Move C:\abc.txt D:\`

Here, `Move` is the program or application responsible for moving the file, while `C:\abc.txt` and `D:\` can be termed as command-line arguments, which are passed to the `Move` program at the time of invocation of the program. As the application is invoked from the command line and the arguments are also passed to the application at the command line itself, these are called *command-line arguments*. Just like C++, programs can be written in Java to accept command-line arguments.

```
public static void main (String args[]){
  . . . . . .
} // end of the main() method
```

In this case, each of the elements in the array named `args` (including the elements at position zero) is a reference to one of the command-line arguments, each of which is a string object.

Suppose, you have a Java application, called `sort`, that sorts the lines in a file named `Sort.text`. You would invoke the Sort application as, `java Sort Example.txt`.

When the application is invoked, the runtime system passes the command-line arguments to the application's `main()`method via an array of strings. In the statement above, the command-line argument passed to the Sort application contains a single string, i.e., `Example.txt`. This String array is passed to the `main()` method and it is copied in *args*.

You must be wondering how many arguments you can supply through a command line. As we have discussed in Section 4.10, the number of elements in an array can be obtained from the `length` property of the array. Therefore, in the signature of `main()`, it is not necessary in Java to pass a parameter specifying the number of arguments. Example 4.27 explains the use of command-line arguments.

Example 4.27 Echo Application

```
L1  class Echo {
L2  public static void main(String args[]){
L3    int x = args.length;
L4    for (int i = 0; i< x; i++)
L5    System.out.println(args[i]);
L6    }
L7  }
```

After compiling the program, when it is executed, you can pass the command-line arguments as follows:

```
C>java Echo A B C
```

Output

```
A
B
C
```

Note that there is one space between each of the three arguments passed to the `Echo` application through the command line. If you have to pass a string of characters as an argument, then you must use quotes (" ") to mark that string. For example,

```
C>java Echo "A is first alphabet" "B is second" "C is third"
```

Output

```
A is first alphabet
B is second
C is third
```

Explanation

L2 `main()`is declared, with an array variable, `args`, referring to an array of strings, passed as command line arguments to the program.

L3 An integer-type variable, `x` is declared to hold the length of the array.

L4–5 `for` loop is iterated from 0 to the length of the array and the value obtained from each iteration is printed in a separate line.

4.12 NESTED CLASSES

Nested class is a class within a class. Nested classes are of the following types:

- Non-static inner classes
- Static nested classes
- Local classes
- Anonymous classes

4.12.1 Inner Class

A *non-static inner class* is a member of the outer class declared outside the functions within a class. The non-static inner class is bound to the instance of the enclosing class and has access to all the members of the enclosing class even the parent's `this` reference and `private` members. An inner class can be defined as `private, default, protected, public, final` and even `abstract`. Each instance of an inner class has a reference to an enclosing outer instance. A reference to the outer class instance can be explicitly obtained through `OuterClassName.this`. You cannot have static variables or methods in an inner class except for compile-time constant variables, i.e., static constant. Inner class's objects can be created within instance methods, constructors of the outer class or through an instance of outer class as they must have a reference to the instance of the outer class. This would be evident from the following example as well. Let us take an example to show how non-static inner classes can be created and used.

Example 4.28 Inner Classes

```
L1  class InnerClassTest
    {
```

Inheritance

Property left to a child may soon be lost; but the inheritance of virtue—a good name an unblemished reputation—will abide forever. If those who are toiling for wealth to leave their children, would but take half the pains to secure for them virtuous habits, how much more serviceable would they be. The largest property may be wrested from a child, but virtue will stand by him to the last. **William Graham Sumner**

After reading this chapter, the readers will be able to

- know the difference between inheritance and aggregation
- understand how inheritance is done in Java
- learn polymorphism through method overriding
- learn the keywords: super and final
- understand the basics of abstract class
- understand the difference between shadowing and overriding

5.1 INHERITANCE VS AGGREGATION

Inheritance, in real life, is the ability to derive something specific from something generic. For example, Fiat Palio parked next to a shopping mall is a specific instance of the generic category, car.

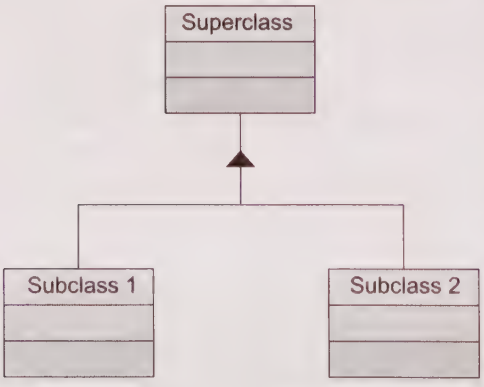


Fig. 5.1 Inheritance

Inheritance aids in the reuse of code, i.e., a class can inherit the features of another class and add its own modification. The parent class is known as the *superclass* and the newly created child class is known as the *subclass*. A subclass inherits all the properties and methods of the super class, and can have additional attributes and methods as shown in Fig. 5.1.

On the other hand, the term aggregation is used when we make up objects out of other objects. The behavior of the bigger object is defined by the behavior of its component objects separately and in conjunction with each other. For example, cars contain an engine which in turn

5.1.2 Deriving Classes Using extends Keyword

In Java, classes are inherited from other class by declaring them as a part of its definition, as shown below.

```
class MySubClass extends
{
...
}
```

In the definition above, the keyword `extends` declares that `MySubClass` inherits the parent class `MySuperClass`.

Now suppose you need a class for a bike or a car. A bike or a car will have a model name, a model year, a maximum speed, a weight, a price, and other characteristic features, but these two will differ in some aspects like a car will possess doors, whereas a bike will not. The properties that are similar can be abstracted and put in a generic class having common behavior. This generic class will be the parent class for both these classes, as shown in Fig. 5.8.

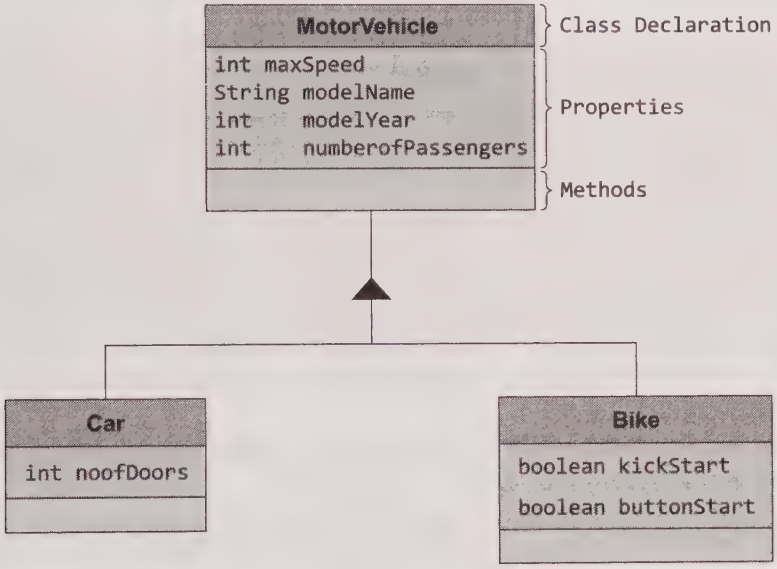


Fig. 5.8 UML Notation for `Car` and `Bike` Class Along with their Parent Class `MotorVehicle`

Now let us frame classes for the above diagram and see how inheritance is actually done in Java. First of all, let us frame the parent class `MotorVehicle`, shown in Example 5.1.

Example 5.1 Parent Class `MotorVehicle`

```
L1  class MotorVehicle{
L2  int maxSpeed;              // miles per hour
L3  String modelName;          // e.g. "Fiat"
L4  int modelYear;             // e.g. 2006,2007,2008
L5  int numberOfPassengers;    // 2, 4, 6

      // we can add some more properties, as above, like the
      // engineCapacity etc. but we would leave it as an
```

```
      // exercise for you.
      // constructor
L6    MotorVehicle()
      {
            maxSpeed = 200;
            modelName = "";
            modelYear=1997;
            numberOfPassengers=2;
      }

L7    MotorVehicle(int maxSpeed, String modelName, int modelYear, int numberOfPassengers)
      {
L8    this.maxSpeed = maxSpeed;
L9    this.modelName = modelName;
L10   this.modelYear = modelYear;
L11   this.numberOfPassengers = numberOfPassengers;
      }
}
```

Explanation

L1 Class `MotorVehicle` has been declared.

L2–5 Instance variable `maxSpeed` (of type `int`), `modelName` (of type `String`), `modelYear` (of type `int`) and `numberOfPassangers` (of type `int`) are declared in these lines.

L6 Default constructor.

L7 Parameterized constructor declaration to initialize the instance variables declared in L2 to L5.

L8–11 The instance variables declared in L2 to L5 are being initialized with the arguments passed in the constructor. The keyword `this` has been used, as the name of both the instance variable and the local variable (arguments in the constructor declaration) are same.

Now let us frame the subclasses, as shown in Fig. 5.8. The example below shows one of the subclasses, i.e., `Bike`.

The subclass `Bike` will have all the features that its parent class possesses. In addition to that, it can have its own features, as shown in Example 5.2.

Example 5.2 Subclass `Bike`

```
L1    class Bike extends MotorVehicle {
L2    boolean kickStart;
L3    boolean buttonStart;
      /* A kick start bike may or may not be a button start bike but a button start bike
      will always have an option of kick starting */
      // constructor
L4    Bike()
      {
         kickStart = true;
         buttonStart = false;
      }

L5    Bike(boolean ks, boolean bs)
      {
         kickStart = ks;
         buttonStart = bs;
```

```
    }
L6  public static void main(String args[]) {
L7  Bike b = new Bike ();
    }}
```

Explanation

L1 Usage of `extends` keyword to show inheritance.
L2 and 3 Declaration of two boolean variables: `kickStart` and `buttonStart`.
L4 Default constructor.
L5 Overloaded constructor.
L6 `main()` method.
L7 `Bike` object is created and the default constructor of the parent class is called first of all and after that, the subclass constructor is called because the parent needs to be initialized before the child.

5.2 OVERRIDING METHOD

When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. Like overloading, it is a feature that supports polymorphism.

When an overridden method is called from within a subclass, it will always refer to the version of the method defined by the subclass. The superclass version of the method is hidden.

In the code below, we see that subclass `B` overrides the method `doOverride()` in class `A`.

Example 5.3 Method of Overriding

```
L1  class A {
L2  int i = 0;
L3  void doOverride (int k) {
L4  i = k;
    } }

    // Subclass definition starts here
L5  class B extends A {            // Method Overriding
L6  void doOverride(int k){
L7    i = 2 * k;
L8    System.out.println("The value of i is: " +i);
      }
L9  public static void main (String args[])
    {
L10   B b = new B();               // Create an instance of class B
L11   b.doOverride(12);            // class B method doOverride() will be called
    }}
```

Output

```
The value of i is: 24
```

Explanation

L1 Class declaration.
L2 Instance variable declaration.
L3 Method declaration with an integer argument passed to it.
L4 Instance variable being assigned with the value of the arguments.
L5 Subclass declaration.
L6 Method is overridden, as the name and signature of the method match.
L7 The variable `i` is being initialized with a value twice to that of the argument passed in the method `doOverride`. Also note that we have not declared the variable `i` in the subclass `B`, it is the parent class variable `i` that is being referred to in the subclass. This is actually what inheritance is all about. Objects of the subclass need not define their own definition of data and methods which are generic in nature. The generic behavior is left for the super classes.
L8 Print statement.
L9 `main()`method.
L10 Object of the subclass `B` is created.
L11 When we create an instance of class `B`, an invocation of the method `doOverride()` will result in a cali to the `doOverride()` code in class `B` rather than `A` because it is actually the instance that matters when we call any instance method and the instance in this case is `B`.

A superclass reference variable can be assigned a subclass object. This is illustrated by the following code:

```
A a1 = new B ();
/* Create an instance of class B but uses reference of type A . */

a1.doOverride();
/* Though the A type reference is used, the doOverride() method of
class B will be called. */
```

Note Remember when you write two or more classes in a single file, the file will be named upon the name of the class that contains the main method. For example, if you write the complete Example 5.3 in a notepad editor file, the file will be named as `B.java`.

Here we see that even though the superclass type variable `a1` references the subclass object, the subclass's overridden method will be executed rather than the superclass's instance method.

This is very useful when, for example, an array of the superclass type contains references to various subclasses. The overridden method in the subclass will be called rather than the method of the superclass.

Code Snippet 5.1 Superclass and Subclass Object

```
L1  A a[] = new A[2];       // Parent Class A type array of two objects
L2  a[0] = new B();         // A type reference is assigned an instance of B.
L3  a[1] = new C();         // A type reference is assigned an instance of C.
L4  for (int i = 0; i<a.length; i++)
L5  a[i].doOverride();
```

Explanation

Considering `A` as the superclass of class `B` and `C`.

L1 An array of superclass `A` is defined with a size of two elements.

L2 The first element of the array is assigned an object of class `B`. An array of `int` contains integers; an array of characters contains characters in its various elements, and so on. An array of superclass `A` will either contain objects of type `A` or its subclasses because *a superclass variable can refer to a subclass object*. So basically, it is an array of objects.

L3 The second element of the array contains the object of class `C`.

L4 `for` loop is used to iterate through various array elements. Value of `i` will vary from 0 to `a.length` which is 2.

L5 In the first iteration, the value of `i` is 0, so `a[0]` refers to `B` class object, so `B`'s `doOverride()` method will be executed even though the array is of the superclass `A`, the code used for the `doOverride()`method will be that of the actual object that is referenced, not of the method in the base class `A`. The same applies for the second iteration. The only difference with the second iteration is that the object will be of class `C`. The reason why subclass methods are being invoked is because these methods are overridden and overridden methods are dynamically binded. Dynamic Binding occurs at runtime and methods are called based on the object from which they have been invoked.

Note Binding is the process of connecting a method call to its body. When binding is performed before a program is executed, it is called *early binding*. When multiple methods with the same name exist within a class (i.e., case of method overloading) which method will be executed depends upon the argument (number, type or order of arguments) passed to the method. So, this binding can be resolved by the compiler (at compile time) and hence overloaded methods are early binded.

When a method with the same name and signature exists in superclass as well as subclass (i.e., a case of Method overriding) which method will be executed (superclass version or subclass version) will be determined by the type of object from which it has been called (Example 5.3) and so it cannot be done by compiler. Objects exist at runtime, and hence *late binding* is done by the JVM at runtime for resolving which overridden method will be executed. It is also known as *dynamic binding* or *runtime binding*.

A superclass reference variable can refer to a subclass object but vice versa is not possible because a superclass can have many subclasses and all of these subclasses can have their additional (different) members (fields and methods) not present in the superclass and its peer classes. Hence a variable of type subclass can expose more details and can perform more operations than a variable of type superclass. So every superclass can refer to its subclass object but every subclass cannot refer to its superclass object. Consider the case of `Furniture` class and its subclasses `Table` and `Chair`. We can say that every table or chair is furniture but we cannot ascertain that all furniture is table or chair, etc. Consider another example of animal and its subclasses like elephant, tiger, dog, cat, etc. Tiger is an animal, elephant is an animal but we cannot say that animal is a tiger or elephant because it is not true in all cases. Let us take an example to see how fields are accessed when we refer to subclass objects using a reference variable of superclass.

Example 5.4 Superclass Can Refer to a Subclass Object

```
L1    class SuperClass
      {
L2          int instanceVariable = 10;
L3          static int classVariable = 20;
      }

L4    class SubClass extends SuperClass
      {
L5      int instanceVariable = 12;
L6      static int classVariable = 25;
L7      public static void main(String args[])
        {
L8        SuperClass s=new SubClass();
L9        System.out.println("Superclass Instance variable: "+s.instanceVariable);
L10       System.out.println("Superclass static variable: "+s.classVariable);
L11       SubClass st=new SubClass();
L12       System.out.println("Subclass Instance variable: "+st.instanceVariable);
L13       System.out.println("Subclass static variable: "+st.classVariable);
        }
      }
```

Output

```
D:\javabook\programs\chap 4>java SubClass
Superclass Instance variable: 10
Superclass static variable: 20
Subclass Instance variable: 12
Subclass static variable: 25
```

Explanation

L1 Class declaration.
L2 Instance variable defined.
L3 Class variable defined.
L4 Subclass declaration.
L5 Instance variable of the subclass has been declared with the same name as that of superclass (shadowing).
L6 Class variable of the subclass has been declared the with same name as that of superclass (shadowing).
L7 `Main` method declaration.
L8 A reference variable `s` of superclass is declared to hold an object of subclass.
L9 The instance variable is printed using `s` (created in L8). The value that is printed (see output) will be of superclass as the reference is of superclass. This binding is made by the compiler at the compile time which checks whether the instance variable belongs to class `Superclass` through which it is being accessed and if yes the binding is made, no matter which object the reference refers to.
L10 Same as L9. The only difference is it is for class variables.
L11 A reference variable `st` of subclass is declared to hold an object of subclass.
L12 The instance variable is printed using `st` (created in L11). The value that is printed (see output) will be of subclass as the reference is of subclass. As already stated, this binding is made by the compiler at the compile time which checks whether the instance variable belongs to class `Subclass` through which it

is being accessed and if yes, the binding is made, no matter which object the reference refers to.

L13 Same as L12. The only difference is it is for class variables.

In the following topics, we will revisit *method overriding* combined with some new topics.

5.3 super KEYWORD

The `super` keyword refers to the parent class of the class in which the keyword is used. It is used for the following three purposes:

1. For calling the methods of the superclass.
2. For accessing the member variables of the superclass.
3. For invoking the constructors of the superclass.

Case 1: Calling the Methods of the Superclass

`super.<methodName>()` represents a call to a method of the superclass. This call is particularly necessary while calling a method of the superclass through the subclass object that is overridden in the subclass.

Example 5.5(a) Simple Example Showing Method Overriding

```
L1   class A {
L2   void show()
       {
L3           System.out.println("Superclass show method");
     }}

L4   class B extends A { // Method Overriding
L5   void show()
       {
L6           System.out.println("Subclass show method");
       }
L7     public static void main (String args[]){
L8     A s1 = new A();   // call to show method of Superclass A.
L9     s1.show();
L10    B s2 = new B();
L11    s2.show();          // call to show method of Subclass B
       }}
```

Output

```
Superclass show method
Subclass show method
```

Explanation

As discussed earlier in Example 5.3, methods will be called on the basis of the objects from which they are called.

L8 Shows the creation of an object of class `A`.

L9 Shows calling the methods of class `A` through the object created in L8.

L10 and 11 The object being used is that of class `B`.

Problem and Solution In Example 5.5, two methods (`show()`and overridden `show()`) are being called by two different objects (`A` and `B`), instead the job can be done by one object only, i.e., by using the keyword `super`. Example 5.5 can be reshaped as shown below:

Example 5.5 (b) Usage of super Keyword for Calling Parent Class Methods

```
L1   class ANew {
L2   void show()
       {
L3     System.out.println("Superclass show method");
       }
       }
L4   class BNew extends ANew { // Method Overriding
L5   void show()
     {
L6     super.show();        //call to show method of the super class A
L7     System.out.println("Subclass show method");
     }
L8   public static void main (String args[]) {
L9     BNew s2 = new BNew();
L10    s2.show();           // call to show method of Subclass B
     }}
```

Output

```
Superclass show method
Subclass show method
```

Explanation

L5 Method `show()` is defined.

L6 Shows how `super` is used for calling the parent class method which has been overridden in the subclass. If this line is omitted, only the subclass method will be called; we have used `super` in this line so that both the version (parent and subclass) of methods can be called by one object only.

L9 `BNew` object is created.

L10 The method `show()`of class `BNew` is called using the object created in L9. The control passes to `show()`in L5 and the statements within the method are executed. L6 gets executed which is `super.show()` and the control passes to `ANew.show` method. Lines of `show()`in `Anew` class are executed and the control passes back to L7, which is a print statement. After executing the print statement (L7), the control passes back to the main method which has no more statements to execute, so the program automatically terminates.

Case 2: Accessing the Instance Member Variables of the Superclass

Example 5.6 demonstrates how the keyword `super` can be used to access the instance variables of the superclass. The keyword is particularly useful when the variable of the superclass is shadowed by the subclass variable. The concept of *shadowing* occurs when variables in super class and subclass have same name. The superclass variables in this case will be hidden in the subclass and only subclass variables will be accessible within the subclass. To access the shadowed variable of superclass `super` keyword is used as shown below.

Example 5.6 Usage of super Keyword for Accessing Parent Class Variables

```
L1   class Super_Variable {
L2   int b = 30;     //instance Variable
     }
L3   class SubClass extends Super_Variable {
L4   int b = 12;     // shadows the superclass variable
L5   void show()
     {
L6     System.out.println("subclass class variable:" + b);
L7     System.out.println("superclass instance variable:" + super.b);
     }
L8   public static void main (String args[]) {
L9     SubClass s = new SubClass();
L10    s.show();     // call to show method of Subclass B
     }}
```

Output

```
subclass class variable: 12
superclass instance variable: 30
```

Explanation

L1 Superclass declaration `Super_Variable`.

L2 Instance variable declaration band it is assigned the value 30.

L3 Subclass declaration `SubClass`.

L4 Instance variable `b` (same name as that of superclass instance variable) within the subclass defined and assigned the value 12.

L5 `Show ()` method defined within the subclass.

L6 and 7 In the above example, specifically we have kept the names of two instance variables in the super and subclass same, i.e., `b`. In L6 when we print the value of `b` by simply writing `b`, the value of subclass variable `b` is printed. In L7, with the help of `super` keyword, we have accessed the value of the superclass instance variable `b` and in this case, it prints the value of the superclass variable. If the variable `b` in L4 is not defined, then both print statements would have printed the same value, i.e., value of the superclass variable `b`. In our case, both the super and the subclass contain the variable with the same name, so to differentiate between the two and to access the value of the superclass variable from the subclass, we use the keyword `super`.

Case 3: Invoking the Constructors of the Superclass

`super` as a standalone statement (i.e., `super()`) represents a call to a constructor of the superclass. This call can be made only from the constructor of the subclass and that too it should be the first statement of the constructors. The default constructors implicitly include a call to the super class constructor using the `super` keyword.

Example 5.7 Constructor Calling Mechanism

```
L1   class Constructor_A {
L2   Constructor_A()
       {
L3     System.out.println("Constructor A");
```

```
    }}
L4  class Constructor_B extends Constructor_A {
L5  Constructor_B() {
L6    System.out.println("Constructor B");
      }}

L7  class Constructor_C extends Constructor_B {
L8  Constructor_C() {
L9    System.out.println("Constructor C");
      }

L10 public static void main (String args[]) {
L11   Constructor_C a = new Constructor_C();
      }}
```

Output

```
Constructor A
Constructor B
Constructor C
```

Explanation

L1 Parent class declaration `Constructor_A`.
L2 Default constructor of class `Constructor_A`.
L4 Subclass declaration `Constructor_B` of the class defined in L1.
L5 Default constructor of class `Constructor_B`.
L7 Subclass declaration `Constructor_C` of class defined in L4.
L8 Default constructor of class `Constructor_C` is declared explicitly.
L10 `main` method declaration.
L11 An object of class `Constructor_C` is created here. If the class does not provide any constructor, the default constructor (no argument constructor) provided by Java is implicitly called when an object of the class is created. All three classes define their respective no argument constructors. Object creation of `Constructor_C` class results in the explicit default constructor of this class being called on L8. An `<init>` method is created for every constructor for the class and this `<init>` includes a call to the superclass default (no argument) constructor, any instance variable initializer provided in the class followed by the code written in the constructor. So when an object of class `Constructor_C` is invoked, the superclass constructor is invoked automatically. Also its parent, i.e., `Constructor_B` needs to be initialized before the child class can be initialized and instantiated. The same case applies for `Constructor_B` which in itself is inherited from `Constructor_A`. Therefore, first of all, the constructor of class `Constructor_A` gets executed, then `Constructor_B` and lastly, `Constructor_C`.

Example 5.7 (a) Usage of super Keyword for Calling Parent Class Constructor

```
L1  class Constructor_A_Revised {
L2  Constructor_A_Revised()
    {
L3    System.out.println("Constructor A Revised");
      }}
L4  class Constructor_B_Revised extends Constructor_A_Revised {
      // this constructor is commented
      /* Constructor_B_Revised() {
      System.out.println("Constructor B");
```

```
    }
    */

L5  Constructor_B_Revised(int a)
        {
        a++;
L6      System.out.println("Constructor B Revised " +a);
        }}
L7  class Constructor_C_Revised extends Constructor_B_Revised {
L8  Constructor_C_Revised()
        {
L9      super(11);      // if omitted compile time error results
L10     System.out.println("Constructor C Revised");
        }
L11 public static void main (String args[]){
L12 Constructor_C_Revised a = new Constructor_C_Revised();
    }}
```

Output

```
Constructor A Revised
Constructor B Revised 12
Constructor C Revised
```

Explanation

(Only the changes are being explained)

L5 The parameterized constructor of class `Constructor_B_Revised` is defined with an integer argument.

L6 The integer argument is being post incremented.

L9 `super` keyword for calling constructor, followed by the argument to be passed to the parent class constructor.

L12 An object of `Constructor_C_Revised` is created due to which the default constructor of this class will be invoked. But as already explained, it is inherited, so its parent's (i.e.,`Constructor_B_Revised`) default constructor will be called automatically. But instead of the default constructor in `Constructor_B_Revised`, a parameterized constructor is provided. If a class does not provide any constructor (default or parameterized), it will be provided with an implicit default constructor automatically by Java. In case the class does provide a constructor, Java will not provide it with a default constructor. An implicit call to the parent class default constructor of `Constructor_C_Revised` results in an error, because the default (no argument) constructor is neither provided nor it will be implicitly available through Java, as a parameterized constructor is provided in the class `Constructor_B_Revised`.

The solution for this is either to explicitly provide a default (no argument) constructor in `Constructor_B_Revised` (shown in comments) or use `super` in the constructor of the subclass `Constructor_C_Revised` (as shown in L9) for making an explicit call to the parameterized constructor in its immediate superclass and in this case, the compiler will not show you an error. The constructor of class `Constructor_A_Revised` is normally called as the default constructor is provided in the class.

Note It is mandatory for a `super` statement in a constructor to be the first statement within the constructor. As the parent must be initialized before its child, an explicit call to the parent must be done before any initialization within the child constructor begins.

5.4 final KEYWORD

The keyword `final` is used for the following purposes:

1. To declare constants (used with variable and argument declaration)
2. To disallow method overriding (used with method declaration)
3. To disallow inheritance (used with class declaration)

Basically, it is used to prevent inheritance and create constants. Let us take an example.

Example 5.8 Final Keyword

```
L1   class Final_Demo {
L2   final int MAX = 100;    //constant declaration
     // final mOethod declaration with final arguments
L3   final void show(final int x) {
L4   // MAX++; illegal statement as MAX is final
L5   // x++; illegal statement as x argument is final
L6   System.out.println("Superclass show method:" +x);
     }}
L7   class Final_Demo_1 extends Final_Demo {
     // cannot override show method as it is final in
     // parent class,that is why we have commented it

L8   /* void show(){
         System.out.println("Subclass show method");
     }*/

L9   public static void main (String args[]){
L10  Final_Demo_1 f2 = new Final_Demo_1();
     //show of the parent class will be called
L11  f2.show(12);
     }}
```

Output

```
C:\examples\> java Final_Demo_1
Superclass show method: 12
```

Explanation

L1 Class declaration `Final_Demo`.

L2 Integer constant declaration `MAX` with value 100.

L3 The `final` method `show()` is defined with final arguments. This method cannot be overridden in its subclasses as is shown in the comments in L8. The final argument's value cannot change, as it has become a constant now (shown in L5).

L7 Subclass declaration. If the parent class would have been a `final` class, then this class could not have been subclassed. The `final` class can be declared as follows:

```
final class Final_Demo
```

5.5 ABSTRACT CLASS

The literary meaning of *abstract* is — "a concept or idea that is not associated with any specific instance." Abstract classes adopt this very concept. Abstract classes are classes with a generic concept, not related to a specific class. They define the partial behavior and leave the rest for the subclasses to provide.

Abstract classes contain one or more abstract methods. It does not make any sense to create an abstract class without abstract methods, but if done, the Java compiler does not complain about it. An abstract method is a method that is declared, but contains no implementation, i.e., no body.

Abstract classes cannot be instantiated, and they require subclasses to provide implementation for their abstract methods by overriding them and then the subclasses can be instantiated. If the subclasses do not implement the methods of the abstract class, then it is mandatory for the subclasses to tag itself as *abstract*, making way for its own subclasses to override the abstract methods.

Why do We Create Abstract Methods?

We use abstract methods, when we want to force the same *name and signature pattern* in all the subclasses and do not want to give them the opportunity to use their own naming patterns, but at the same time give them the flexibility to code these methods with their own specific requirements. Example 5.9(a) shows an abstract `Animal` class. This class has been specifically created as an abstract class due to the presence of abstract methods in it. There are certain features that are common to all the animals but certain other features are specific to a category of animal. We may also argue that the common features are performed in a variety of ways by different animals. For example, every animal in this world produce a particular kind of sound, unique to their own species.

Example 5.9 (a) Abstract Class with Abstract Method

```
L1  abstract class Animal
    {
L2    String name;
L3    String species;
    // constructor of the abstract class
L4    Animal(String n, String s)
      {
L5    name = n;
L6    species = s;
      }
L7   void eat(String fooditem)
      {
L8    System.out.println(species + " " + name + " likes to have " + fooditem);
        }
L9   abstract void sound();
    }
```

Explanation

L1 Abstract class declared with the keyword `abstract` used before the class declaration.
L2 and 3 Two string variables declared, named `name` and `species`.
L4 Parameterized constructor to initialize the instance variable.
L5 and 6 Instance variables, `name` and `species`, are initialized with the arguments passed to the constructors in L4.
L7 A non-abstract method has been defined, just like other normal methods.
L8 Print statement.
L9 Abstract method declared. Note that this method does not have any body.

The `abstract` keyword is used for defining both abstract methods and abstract classes. Any animal that wants to be instantiated must override the `sound()`method, otherwise it is impossible to create an instance of that class. Let us take a look at the `Lion` subclass that inherits the `Animal` class.

Example 5.9 (b) Class Implementing Abstract Methods

```
L1   class Lion extends Animal
     {
L2   Lion() {
L3     super("Lion","Asiatic Lion");
     }
L4   void sound() {
L5     System.out.println ("Lions Roar! Roar!");
       }
L6   public static void main(String args[])
     {
L7     Lion l = new Lion();
L8     l.eat("flesh");
L9     l.sound();
     }}
```

Output

```
Asiatic Lion likes to have flesh
Lions Roar! Roar!
```

Explanation

L1 Subclass declaration of the abstract class `Animal`.
L2 Default constructor created for `Lion` class.
L3 The keyword `super` used to set up an explicit call to the parent class constructor.
L4 It is mandatory for the subclass `Lion` to override the `sound()` method because the `sound()`method has been declared abstract by the parent class.
L7 The object of `Lion` class is created.
L8 The `eat()`method (Example 5.9(a)) of the parent class will be called with the help of the object created
in L7.
L9 The `sound()` method is called which has been declared in L4.

Some key features of an abstract class are as follows:

1. They cannot be instantiated, but they can have a reference variable.
2. A class can inherit only one abstract class, as multiple inheritance is not allowed amongst classes.
3. They can have abstract methods as well as non-abstract methods.
4. It is mandatory for a subclass to override the abstract methods of the abstract class, otherwise the subclass also need to declare itself as abstract. Overriding other methods (non-abstract) is up to the requirement of the subclass.
5. Abstract classes can have constructors and variables, just like other normal classes.

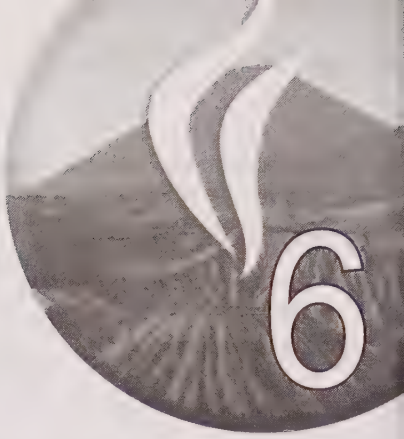
5.6 SHADOWING VS OVERRIDING

Shadowing of fields occurs when variable names are same. It may occur when local variables and instance variable names collide within a class or variable names in superclass and subclass are same. In case of methods, instance methods are overridden whereas static methods are shadowed. The difference between the two is important because shadowed methods are bound early whereas instance methods are dynamically (late) bound. The difference is illustrated in the following example.

Example 5.10 Shadowing vs Overriding

```
L1   class Shadowing
     {
L2     static void display()
       {
L3           System.out.println("In Static Method of Superclass");
       }
L4     void instanceMethod()
       {
L5           System.out.println("In instance Method of Super Class");
       }
     }

L6     class ShadowingTest extends Shadowing
       {
            // Static Methods are not Overridden but Shadowed
L7          static void display()
            {
L8          System.out.println("In Static Method of Sub Class");
            }
       // instance methods are Overridden not shadowed
L9          void instanceMethod()
            {
L10           System.out.println("The Overridden instance Method in Sub Class");
            }
L11    public static void main(String args[])
```



Interfaces, Packages, and Enumeration

The greater our knowledge increases, the more our ignorance unfolds.

John F. Kennedy

After reading this chapter, the readers will be able to

- understand what interfaces are and how they are different from abstract classes
- understand the concept behind packages and how they are used
- know about the `java.lang` package
- understand object class and wrapper class
- know how strings are created, manipulated, and split in Java
- understand enumerations

6.1 INTERFACES

Interfaces in Java are like a contract or a protocol which the classes have to abide with. Interfaces are basically a collection of methods which are public and abstract by default. These methods do not have any body. The implementing objects have to override all the methods of the interface and provide implementation for all these methods. There is no code at all associated with any method of the interface. The best part of an interface is that a class can inherit any number of interfaces, thus allowing *multiple inheritance* in Java, provided the class now has to override all the methods of all the interfaces it inherits. Java does not support multiple inheritance among classes, but interfaces allow Java to support this feature.

Interfaces are declared with the help of a keyword `interface`. Note that none of the methods have a body. It is the responsibility of the implementing class to override the methods and provide the implementation for these methods.

```
interface interfacename
{
     returntype methodname(argumentlist);
     ...
}
class classname implements interfacename{}
```

Example 6.1(a) shows a very simple calculator program. There are a few basic operations that do not change for any calculator: be it a normal, scientific, or a programmable calculator. The basic operations (add, subtract, divide, and multiply) can be squeezed out of various implementing classes and put into an interface. Now all the implementing objects will have to keep the name and signature of the methods exactly same as has been defined in the interface, that is why we have created an interface and this is what we actually wanted for all the subclasses to follow. We do not want the classes to follow their own set of rules like their own created method names and their signatures. We wanted the classes to follow the rules set up by the interfaces and it will be a binding upon them, but these rules will be implementation independent. That is, the objects have to code according to their own requirement within the overridden methods. For simplicity, we have created an interface named `Calculator` and four methods have been defined in it to denote four basic operations of a calculator and these methods perform operations only on integers. You can later on extend this program to accept different kinds of arguments such as `double`, `float`, and `byte`.

Classes, while inheriting other classes, use the keyword `extends`; whereas while inheriting an interface, they use the keyword `implements`, as shown in Example 6.1(b).

Example 6.1 (a) `Calculator.java`: Interface Definition

```
L1    interface Calculator
      {
L2      int add(int a,int b);
L3      int subtract(int a,int b);
L4      int multiply(int a,int b);
L5      int divide(int a,int b);
      }
```

Explanation

L1 The keyword `interface` has been used to declare an interface followed by the name of the interface and opening curly brackets to denote the starting of interface.

L2 A method named `add` has been declared with the return type `int` that accepts two arguments of type `int`.

L3 A method named `subtract` has been declared with the return type `int` that accepts two arguments of type `int`.

L4 A method named `multiply` has been declared with the return type `int` that accepts two arguments of type `int`.

L5 A method named `divide` has been declared with the return type `int` that accepts two arguments of type `int`, followed by the closing curly bracket of the interface.

Example 6.1 (b) `Normal_Calculator.java`: Class Implementing Calculator Interface

```
L1    class Normal_Calculator implements Calculator
        {
L2      public int add(int a,int b){
L3        return a + b; }
L4      public int subtract(int a,int b) {
L5        return a - b; }
L6      public int multiply(int a,int b) {
L7        return a * b; }
L8      public int divide(int a,int b)
```

```
    {
L9      return a / b;
     }
L10  public static void main(String args[]) {
L11  Normal_Calculator c = new Normal_Calculator();
L12  System.out.println("Value after addition = "+c.add(5,2));
L13  System.out.println("Value after Subtraction = " +c.subtract(5,2));
L14  System.out.println("Value after Multiplication = " +c.multiply(5,2));
L15  System.out.println("Value after division = " +c.divide(5,2));
    }}
```

Output

```
C:\javabook>java Normal_Calculator
Value after addition = 7
Value after Subtraction = 3
Value after Multiplication= 10
Value after division = 2
```

Explanation

L1 Class `Normal_Calculator` has been declared and it inherits the interface `Calculator` with the help of *implements* keyword.

L2 Method `add` has been overridden and the body of the method has been provided.

L3 The keyword `return` is used to return the result (to the caller) of addition of two arguments passed into the add method followed by the closing curly bracket.

L4–9 The methods `substract`, `multiply`, and `divide` are overridden and the results are returned.

L11 An object of the class `Normal_Calculator` is created.

L12–15 Print statements to print the result of addition/subtraction/multiplication/division. Respective methods have been called in these lines with the object created in L11 like `c.sum(5,2)`. These method calls return the result and the result is concatenated with the strings passed as an argument to the `println` method and displayed on the screen (see output).

Note It is mandatory to add the access specifier `public` to the method declaration, otherwise the compiler will not compile the program. As already discussed, all the methods in the interface are public, so when the implementing classes override the methods defined in the interface, they have to tag it as public.
Not making it public or leaving the access specifier blank (default) will reduce the privileges from public to default, which is not allowed in overriding. Either you have to increase the privileges or keep it intact. Widening conversion in case of overriding takes place automatically, i.e., from default to public (lesser privileges to more privileges), but narrowing conversion is not allowed.

It is recommended to create two java files in a directory: (a) `Calculator.java` for defining the interface and (b) `Normal_Calculator.java` for declaring the class implementing the `Calculator.java` interface. The compiler upon compilation of `Normal_Calculator.java` will create two class files automatically: `Calculator.class` and `Normal_Calculator.class`.

6.1.1 Variables in Interface

Just like methods in an interface (by default `public` and `abstract`; no need to tag them), variables defined in an interface also carry a default behavior. They are implicitly `public`, `final`, and `static`

and there is no need to explicitly declare them as *public*, *static*, and *final*. As they are `final`, they need to be assigned a value compulsorily. Being `static`, they can be accessed directly with the help of an interface name and as they are `public`, we can access them from anywhere. Example 6.2 shows the usage of variables in an interface.

Example 6.2 Variables in an Interface

```
L1   interface Limit_Test {
L2     int LOWERLIMIT = 0;
L3     int UPPERLIMIT = 100;
       }
L4   class Variable_Test implements Limit_Test {
L5   void findNumberWithinLimits(int a) {
L6   if(a > LOWERLIMIT && < < UPPERLIMIT)
L7     System.out.println(a+ " lie in between" + Variable_Test.LOWERLIMIT + " and " +
       Variable_Test.UPPERLIMIT);
L8   else
L9     System.out.println(a+ " does not lie in between " + Variable_Test.LOWERLIMIT +
       " and " +Variable_Test.UPPERLIMIT);
       }
L10  public static void main(String args[]) {
L11    Variable_Test vt = new Variable_Test();
L12    //LOWERLIMIT++; illegal statement
L13    //UPPERLIMIT++;
L14    vt.findNumberWithinLimits(23);
L15    vt.findNumberWithinLimits(233);
       }}
```

Output

```
C:\javabook\>java Variable_Test
23 lie in between 0 and 100
233 does not lie in between 0 and 100
```

Explanation

L1 We have created an interface `Limit_Test`, wherein we will set the upper and lower limits.

L2 and 3 The upper and lower limits are being set with the help of two variables in the interface, i.e., `UPPERLIMIT` and `LOWERLIMIT`. They have to be assigned a value, as they are implicitly final.

L4 Class `Variable_Test` inheriting the interface `Limit_Test`.

L5 Method `findNumberWithinLimits` is declared with an argument. This argument will be checked by the method whether it is within the limit or not.

L6 A simple `if` condition to check whether the argument passed in the function (L5) is greater than the `LOWERLIMIT` and lesser than the `UPPERLIMIT`. If the condition satisfies, L7 is executed, else L9.

L7 `print` statement to print that the argument lies in between the limits defined by the interface. Note that the static variable's `LOWERLIMIT` and `UPPERLIMIT` have been accessed with the help of the interface `Variable_Test`.

L9 `print` statement to print that the argument does not lie in between the limits defined by the interface (same as in L7).

L11 An object of the class `Variable_Test` is created.

L12 and 13 Commented statements to modify the variables: `LOWERLIMIT` and `UPPERLIMIT`. If uncommented, these statements will result in a compile-time error because the variables defined in an interface are final, and final variable values cannot be modified.

L14 `findNumberWitihinLimits()` is called through the object of `Variable_Test`, and an argument of 23

is passed. This argument is checked by the method to be within the lower limit and the upper limit and if yes, the value is printed on screen.

L15 `findNumberWitihinLimits()` is called through the object of `Variable_Test`, and an argument of 233 is passed. This argument is checked by the method to be within the lower limit and the upper limit and if it is not, print on screen.

6.1.2 Extending Interfaces

Just like normal classes, interfaces can also be extended. An interface can inherit another interface using the same keyword `extends`, and not the keyword `implements`. Example 6.3 shows how interfaces are extended.

Example 6.3 Extending Interfaces

```
L1   interface A {
L2     void showA();
       }
L3   interface B extends A{
L4     void showB();
       }
L5   class InDemo implements B {
L6   public void showA()
              {
              System.out.println("Overriden method of Interface A");
              }
L7   public void showB()
              {
              System.out.println("Overriden method of Interface B");
              }
     public static void main (String args[])
       {
       InDemo d = new InDemo();
       d.showA();
       d.showB();
       }}
```

Output

```
C:\javabook\>java InDemo
Overriden method of Interface A
Overriden method of Interface B
```

Explanation

L1 An interface named `A` has been declared.

L2 Method `showA()` has been defined in interface `A`.

L3 Interface `B` is defined and we have used `extends` in its declaration to indicate that the parent interface of `B` is `A`. Any class that inherits `B` will have to override all the methods of interface `A` as well as `B`.

L4 Method `showB()` has been defined in interface `B`.

L5 Class declaration shows that it inherits the interface `B`.

L6 Shows the overridden method `showA()`. Note that while overriding, `public` access specifier is added.

L7 Shows the overridden method `showB()`.

6.1.3 Interface vs Abstract Class

Table 6.1 lists the differences between interface and abstract class.

Table 6.1 Interface vs Abstract Class

Interface	Abstract Class
Multiple inheritance possible; a class can inherit any number of interfaces.	Multiple inheritance not possible; a class can inherit only one class.
implements keyword is used to inherit an interface.	**extends** keyword is used to inherit a class.
By default, all methods in an interface are **public** and **abstract**; no need to tag it as **public** and **abstract**.	Methods have to be tagged as **public** or **abstract** or both, if required.
Interfaces have no implementation at all.	Abstract classes can have partial implementation.
All methods of an interface need to be overridden.	Only abstract methods need to be overridden.
All variables declared in an interface are by default **public**, **static**, or **final**.	Variables, if required, have to be declared as **public**, **static**, or **final**.
Interfaces do not have any constructors.	Abstract classes can have constructors.
Methods in an interface cannot be static.	Non-abstract methods can be static.

It is not that interfaces and abstract classes are entirely dissimilar, they have some similarities also.

1. Both cannot be instantiated, i.e., objects cannot be created for both of them.
2. Both can have reference variables referring to their implementing classes objects. For example, if `X` is an interface and its implementing class name is `Y`, then we cannot code:

```
X x1 = new X();          // illegal code
```

   But we can code,

```
X x1 = new Y();          // legal code
```

3. Interfaces can be extended, i.e., one interface can inherit another interface, similar to that of abstract classes (using `extends` keyword).
4. `static/final` methods can neither be created in an interface nor can they be used with abstract methods.

6.2 PACKAGES

You must have encountered situations wherein you try to organize too many files in folders/directories and subdirectories. Similarly, if you have too many classes at your disposal, some sort of grouping is required. Java package is one such mechanism for organizing Java classes into groups. In fact, a package is indeed a directory for holding Java files. Java has many such predefined packages which can be used in programs. Some of the predefined packages in Java are `applet`, `awt`, `lang`, `util`, `event`, `io`, `swing`, etc. Programmers are also permitted to develop their own packages in order to organize classes belonging to the same category or providing similar functionality.

Note A package can be defined as a collection used for grouping a variety of classes and interfaces based on their functionality.

It is also possible to house these Java packages, as these can be stored in compressed files called JAR files (a JAR file or Java ARchive is used for aggregating many files into one) allowing classes to download faster as a group rather than one at a time.

A package declaration resides at the top of a Java source file. All source files to be placed in a package have a common package name.

Table 6.1 Interface vs Abstract Class

Interface	Abstract Class
Multiple inheritance possible; a class can inherit any number of interfaces.	Multiple inheritance not possible; a class can inherit only one class.
implements keyword is used to inherit an interface.	**extends** keyword is used to inherit a class.
By default, all methods in an interface are **public** and **abstract**; no need to tag it as **public** and **abstract**.	Methods have to be tagged as **public** or **abstract** or both, if required.
Interfaces have no implementation at all.	Abstract classes can have partial implementation.
All methods of an interface need to be overridden.	Only abstract methods need to be overridden.
All variables declared in an interface are by default **public**, **static**, or **final**.	Variables, if required, have to be declared as **public**, **static**, or **final**.
Interfaces do not have any constructors.	Abstract classes can have constructors.
Methods in an interface cannot be static.	Non-abstract methods can be static.

It is not that interfaces and abstract classes are entirely dissimilar, they have some similarities also.

1. Both cannot be instantiated, i.e., objects cannot be created for both of them.
2. Both can have reference variables referring to their implementing classes objects. For example, if `X` is an interface and its implementing class name is `Y`, then we cannot code:

```
X x1 = new X();          // illegal code
```

   But we can code,

```
X x1 = new Y();          // legal code
```

3. Interfaces can be extended, i.e., one interface can inherit another interface, similar to that of abstract classes (using `extends` keyword).
4. `static/final` methods can neither be created in an interface nor can they be used with abstract methods.

6.2 PACKAGES

You must have encountered situations wherein you try to organize too many files in folders/directories and subdirectories. Similarly, if you have too many classes at your disposal, some sort of grouping is required. Java package is one such mechanism for organizing Java classes into groups. In fact, a package is indeed a directory for holding Java files. Java has many such predefined packages which can be used in programs. Some of the predefined packages in Java are `applet`, `awt`, `lang`, `util`, `event`, `io`, `swing`, etc. Programmers are also permitted to develop their own packages in order to organize classes belonging to the same category or providing similar functionality.

Note A package can be defined as a collection used for grouping a variety of classes and interfaces based on their functionality.

It is also possible to house these Java packages, as these can be stored in compressed files called JAR files (a JAR file or Java ARchive is used for aggregating many files into one) allowing classes to download faster as a group rather than one at a time.

A package declaration resides at the top of a Java source file. All source files to be placed in a package have a common package name.

- A package provides a unique namespace for the classes it contains.
- A package can contain the following:
  - Classes
  - Interfaces
  - Enumerated types
  - Annotations (metadata facility for elements introduced in Java 5)
- Two classes in two different packages can have the same name, which is not possible without using the package mechanism.
- Packages provide a mechanism to hide its classes from being used by programs or packages belonging to other classes.

6.2.1 Creating Packages

Until now, we have studied what packages are and why they are used. The packages in Java can be of two kinds, predefined Java API packages and user-defined packages. Java 6 API has a large number of classes and interfaces, housed according to their functionality into different packages. Some of these are listed in Table 6.2.

Table 6.2 Commonly Used Predefined Packages

Package	Functionality
`java.lang`	Basic language fundamentals
`java.util`	Utility classes and collection data structure classes
`java.io`	File handling operations
`java.math`	Arbitrary precision arithmetic
`java.net`	Network programming
`java.sql`	Java Database Connectivity (JDBC) to access databases
`java.awt`	Abstract window toolkit for native GUI components
`javax.swing`	Lightweight programming for platform-independent rich GUI components

The above-mentioned packages are pre-designed to be a part of Java API. Now the question arises—how can the users create their own packages?

The name of the package should be followed by the keyword `package`, declared at the top of the program. Anything else, say class declaration and so, may only be followed by the package declaration. Thus, we can define a class belonging to a package as follows:

```
package packexample;                    //package declaration
public class ClassinPackage
{
  //class definition inside package
  //Body of class
}
```

Saving, Compiling, and Executing Packages

Here, the package name is `packexample` and the class `ClassinPackage` has been made a part of this package. There are two ways of saving, compiling, and executing Java files stored in a package.

1. Remember the file must be saved with the name of the class, i.e., `ClassinPackage` and placed in the directory named exactly the same as the package, i.e., `packexample`. The file is compiled from within the package and the class file generated after compilation is stored in the same directory (package). For executing the file, move up the current directory and execute the file by mentioning the name of the `package()` followed by the class name. For example, suppose the package `packexample` is within the directory *pack*. The sequence of statements would be:

```
        // compiling the class
L1      C:\pack\packexample\> javac ClassinPackage.java

        // executing the class
L2      C:\pack\> java packexample.ClassinPackage

        // This will not execute as c does not have packexample directory.
L3      C:\> java packexample.ClassinPackage
```

Note Classes that reside inside a package cannot be referred by their own name alone. The package name has to precede the name of the class of which it is a part of. All classes are a part of some or the other package. If the keyword package is not used in any class for mentioning the name of the package, then it becomes a part of the default/unnamed package. In that case, we execute the classes as shown earlier.

2. This Java source file could be saved in any directory. During compilation time, you need to specify an option of the Java compiler *–d* which specifies the destination where you want to place your generated compiled files. After successful compilation, you would see that your package has been already created in your specified path and the `.class file` has been placed in that package. For executing the class, same steps need to be followed as explained above. Let us consider, the `ClassinPackage.java` file is stored in the `javaeg` directory.

```
        //-d option used with javac for specifying destination c:\pack
        // syntax: javac -d destination directory followed by java source file
L1      C:\javaeg\>javac -d c:\pack ClassinPackage.java

        // executing the class /
L2      C:\pack\> java packexample.ClassinPackage

        // This will not execute
L3      C:\> java packexample.ClassinPackage
```

Note In both the cases, the execution takes place from the parent directory of the package where the class files are placed as shown in L2. If we want to execute the package from any of the directories, the `classpath` should be set.

Setting the `Classpath`

`Classpath` is used for storing the path of the third-party and user-defined classes. Whenever we execute/compile any class file, jdk tools `javac` and `java`, search the package/class file in the user classpath which is the current directory by default. If the classes are not in the current directory, then we need to set the `classpath`.

The `classpath` can be set in two ways:

1. It is an environment variable which can be set using the *System* utility in the control panel or at the DOS prompt as shown.

   ```
   Set CLASSPATH = %CLASSPATH%;c:\pack;
   ```

   `%Classpath%` is used to keep the existing path intact and append our new path to it. Now L3 of both the above cases will execute.

Note Setting `classpath` at the DOS prompt will have to be done each time you open the DOS prompt, as closing the prompt resets the `classpath` to its original value. To make the changes permanent, edit the environment variable in the control panel.

Do not delete the existing `classpath`; edit the variable to append your `classpath` to the environment variable.

2. Use `classpath` option `-classpath` or `-cp` of `javac/java` tools to override the user-defined `classpath` and find the user-defined specific package/classes used in the Java source files.

   ```
   //syntax: javac -cp path of the directory/package used in java source file
   followed by name of the java source file
   C:\pack\packexample> javac -cp c:\javaeg DemoClass.java
   ```

   `-cp` specifies that the user-defined package/classes used in `DemoClass.java` will be found at `c:\javaeg`.

Subpackages

Subpackages can be designed in hierarchy, i.e., one package can be a part of another package. This can be achieved by specifying multiple names of the packages at various levels of hierarchy, separated by dots. For example,

```
package rootpackage.subpackage1;
```

As related classes can be collected in a package, related packages can also be collected in a larger package. In the above statement, `subpackage1` is designed to be a part of `rootpackage`. Of course the hierarchical packages have to be stored in a hierarchical structure of directories and subdirectories. For example, the above package `subpackage1` (which is a part of `rootpackage`) will be stored within the directory `rootpackage`.

Note The names of the packages and the directories have to be same, and the names being used should be carefully selected.

6.2.2 Using Packages

In Java, the names of classes that are defined inside various packages can always be referenced by specifying the names of the corresponding packages to which these classes belong to. For example, the `Rectangle` class belonging to the package `java.awt` can be referred to

`java.awt.Rectangle`, i.e.

```
java.awt.Rectangle box = new java.awt.Rectangle (5, 10, 20, 30);
```

But Java provides an import mechanism which can be used and classes can be used without prefixing the names of packages they belong to.

The point worth noting here is that the class `Rectangle` is referred to by preceding it with the name of the package `java.awt`. Certainly, this is a tedious process. The statement given below is a convenient form of the above statement.

```
Rectangle box = new Rectangle (5, 10, 20, 30);
```

This statement will do fine only if you *import* the class beforehand, i.e., at the start of the program itself.

Now the question is how to import the classes belonging to the various packages. Classes in a package like `java.lang` are automatically imported. For all other classes, you must supply an `import` statement to either import a specific class

```
import java.awt.Rectangle;
```

or to import all the classes in a package, using the *wildcard* notation.

```
import java.awt.*;
```

Let us try and implement the things we have discussed till now. In the following example, we have created two packages `packexample` and `packexample1`. The `packexample` has a class `ImportExmaple` which will be used in the class `UseImportExmaple` of another package `packexample1`.

Example 6.4 Recursive Program to Calculate Factorial in a Package

```
L1  package packexample;
L2  public class ImportExample{
L3  public int fact(int a){
L4    if(a == 1)
L5        return 1;
L6    else
L7        return a*fact(a-1);
    }}
```

Explanation

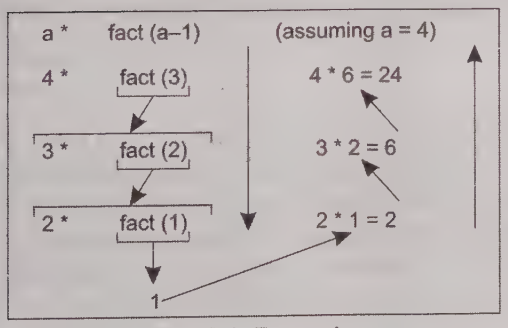


Fig. 6.1 Recursion

L1 Package named `packexample` has been declared.

L2 Public class named `ImportExample` within the package `packexample` has been declared.

L3 Public method `fact` has been defined.

L4 and 5 If the value of `a` is 1, return 1.

L6 and 7 Else return a * call to `fact` method with the argument `a - 1`. A function is being called from within, i.e., recursion. Fig. 6.1 shows the sequence of execution of this recursive function.

Example 6.5 Using a Package in Another Package and Calling the Recursive Factorial Method

```
L1  package packexample1;
L2  import packexample.*;
L3  class UseImportExample{
L4  public static void main(String args[]){
L5    int a = 4;
L6    ImportExample i = new ImportExample();
L7  System.out.println("factorial of " +a+ " is " +i.fact(a)); }}
```

Output

```
factorial of 4 is 24
```

Explanation

L1 Package `packexample1` is defined.

L2 We wanted to access the class `ImportExample` in our class, so we need to import the package of the class `ImportExample`, i.e., `packexample`. If we do not use the import statement, the compiler would complain about using `ImportExample` in L6.

Following are the steps to compile and execute this program:

```
C:\javabook\programs\packexample1>javac -cp c:\javabook\programs
     UseImportExample. java
C:\javabook\programs\packexample1>java -cp c:\javabook\programs;. packexample1.
     UseImportExample
factorial of 4 is 24
```

The-cp option specifies the path of package(directory) from where to access classes used in `UseImportExample.java`. The dot at the end in the classpath has been added to allow the UseImportExample in packexample1 locate itself. The dot represents the current directory. The `programs` directory is the common parent directory of both `packexample` and `packexample1`, so the package `packexample` can be easily imported as the `classpath` is specified using `-cp` option (i.e., `c:\javabook\programs`). There is no problem in locating `packexample` in `packexample1`. `UseImportExample` as classpath for both package is same which has already been given in the commmand.

Note You can also set the `classpath` using the set command at the DOS prompt or set it permanently in the environment variables in the control panel so that you don't have to use the – `cp` option again and again with the JDK tools.

Static Import

In Section 4.7, we have already discussed about the static fields and methods of a class. We invoked the static fields and methods of a class preceding each with the class name and a dot (.). Static import a feature was introduced in Java 5. It enables programmers to use the imported static members as if they were declared in the class itself. The name of the class and a dot (.) are not required to use an imported static member. The following statement shows how to use static import:

```
import static pkgName.[subPkgName].ClassName.staticMemberName;
```

- `pkgName` is the name of the package containing the class whose static members need to be imported.
- `subpkgName` is the name of the subpackage to which the class belongs. The square brackets indicate that it is optional.
- `ClassName` is the name of the class whose static members need to imported.
- `staticMemberName` is the name of the static field or method. But the above statement would import only the mentioned static member of the class.

If you want to import all the static members of the class, then use the following:

```
import static pkgName.ClassName.*;
```

Note Static import imports only static members of the class. Normal import statements should be used to import the classes used in a program.

Example 6.6 demonstrates the use of static import.

Example 6.6 Usage of Static Import

```
L1  import static java.lang.Math.*;
L2  public class ExampleStaticImport {
L3  public static void main(String args[]) {
L4    System.out.println("power of 2 raise to 2 is: " +pow(2,2));
L5    System.out.println("ceil(-10.2)is: " + ceil(-10.2));
L6    System.out.println("floor(-10.2)is: " +floor(-10.2));
L7    System.out.println("ceil(10.2)is: " + ceil(10.2));
L8    System.out.println("floor(10.2)is: " +floor(10.2));
L9    System.out.println("maximum of 23 and 24 is: " +max(23,24));
L10   System.out.println("minimum of 23 and 24 is: " +min(23,24));
L11   System.out.println("value of PI is: " +PI);
L12   System.out.println("Value of E is: " +E);
    }}
```

Output

```
C:\javabook>java ExampleStaticImport
power of 2 raise to 2 is: 4.0
ceil(-10.2)is: -10.0
floor(-10.2)is : -11.0
ceil(10.2)is: 11.0
floor(10.2)is: 10.0
maximum of 23 and 24 is: 24
minimum of 23 and 24 is: 23
value of PI is: 3.141592653589793
Value of E is: 2.718281828459045
```

Explanation

L1 It is a static import declaration that imports all static fields and methods of the class `Math` from the package `java.lang`.

L4–12 Show a few static methods (`pow`, `floor`, `ceil`, `min`, and `max`) and fields PI and E being accessed without preceding the field name or method names with the class name `Math` and a dot as we had used `import static` at the top. If we use normal import instead of static import, then each function and field has to precede with the class name.

6.2.3 Access Protection

Access protection defines how much an element (class, method, variable) is exposed to other classes and packages. There are four types of access specifiers available in Java (shown in the decreasing order of access).

- `public` • applied to variables, constructors, methods, and classes
- `protected` • applied to variables, constructors, methods, and inner classes (not top-level classes)
- `default` • applied to variables, constructors, methods, and classes
- `private` • can be applied to variables, constructor, methods and inner classes (not top-level classes)

`public` and `private` are easy to define. The former means accessibility for all and the latter means accessibility from within the class only. The discussion settles down to two access specifiers: `protected` and `default`. `default` (blank) access specifiers are accessible only from within the package and protected access is beyond the package also but only to the subclasses outside the package. Let us take an example to understand the access specifiers. Assuming

```
x & y     are packages
A         is a public class within package x
B         is another class within package x
C         is subclass of A in package x
D         is subclass of A within package y
E         is class within package y
abc()     is a method with default access in class A
xyz()     is a method with protected access specifier in class A
pqr()     is a method with public privileges in class A
```

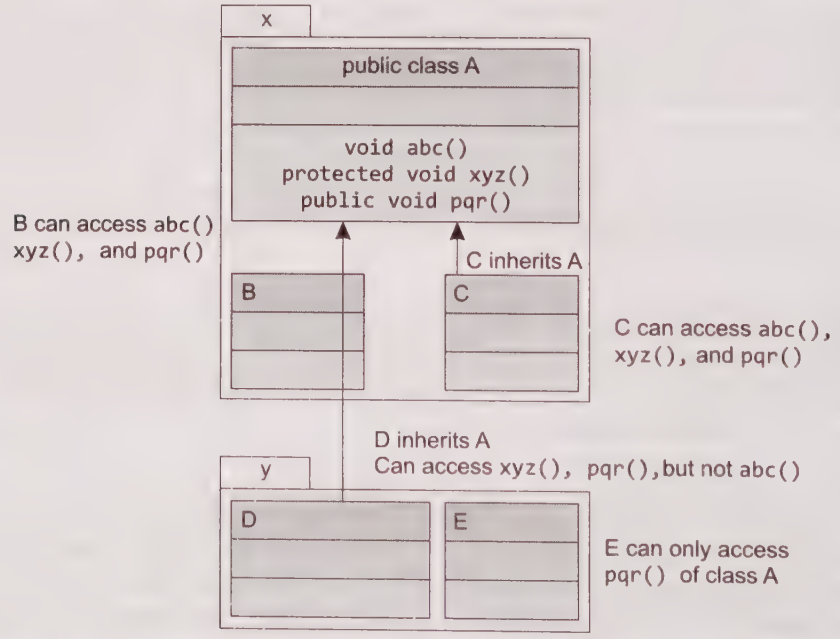


Fig. 6.2 UML Representation of Package and Classes to Show Access Protection

This method `abc()` is accessible from A, B, and C, but neither from D nor E. `protected` methods are also accessible outside the package, but only to the subclasses outside the package. For example, the method `xyz()` is accessible from classes A, B, C, D, but not from E. This is pictorially shown in Fig. 6.2. The method `pqr()` is accessible from all the classes, as it is a public method in a public class.

Note Access of any element, such as variable and method, is also governed by its container. For example, suppose a default access level class has a public method. This method is available to all the classes within the package, but not outside it, as the class in which the method has been defined is not accessible outside the package.

6.3 `java.lang` PACKAGE

`java.lang` is a special package, as it is imported by default in all the classes that we create. There is no need to explicitly import the `lang` package. It contains the classes that form the basic building blocks of Java.

Note Remember we have been using `String` and the `System` class from the first example in this book, but we have not imported any package for using these classes, as both these classes lie in the `lang` package. There are various classes in the `lang` package and it is not possible to discuss all the classes, but we will discuss some of the very important ones.

6.3.1 `java.lang.Object` Class

`Object` class is the parent of all the classes (predefined and user-defined) in Java. For all the classes that we have created so far or will be creating further, `Object` class is the parent by default and there is no need to explicitly inherit the `Object` class. The methods of `Object` class can be used by all the objects and arrays. The method `toString()` and `equals()` have been overridden by many of the predefined classes already. We have already seen what happens when we try to print an object. The `toString()` method is implicitly invoked when an object of any class is printed. If class does not provide a `toString()` method, the `toString()` of the superclass is invoked. If any of the class in the hierarchy does not provide implementation for the `toString` method, the `Object` class method is called. If `toString()` method of the `Object` class is called, `classname@hexadeciaml representation of hash code of the object` is printed. In case you wish to provide your own definitions for the objects which should be returned once you try to print your objects, then you must override the `toString()` method and return your own defined strings for the objects (Table 6.3).

Table 6.3 Methods of the `java.lang.Object` Class

Method	Description
`Object clone()`	A copy of the object is created and returned
`boolean equals(object o)`	Checks whether an object is equal to another or not. If both references refer to the same object, they are equal, else not.
`void finalize()`	Used by classes to dispose of their occupied resources

(Contd)

(Table 6.3 Contd)

Method	Description
`final Class getClass()`	Returns the class of the object
`int hashCode()`	Return the hash code of the object
`final void notify()`	Used by threads, to wake up a thread that is in waiting state
`final void notifyAll()`	Used by threads, to wake up all the threads in waiting state
`String toString()`	A string definition of the object is returned
`final void wait()`	Puts the current thread in waiting state
`final void wait(long time)`	Puts the current thread in waiting state for the specified time
`final void wait(long time, int n)`	Puts the current thread in waiting state for the specified amount of real time

Example 6.7 `toString()` **Method of** `Object` **Class**

```
L1   class Demo {
L2   public String toString()
     {
L3     return "My Demo Object created";
     }
L4     public static void main(String args[])
     {
L5     System.out.println(new Demo());
     }}
```

Output

```
C:\javabook\chap 6>java Demo
My Demo Object created
```

Explanation

L2 `toString()` method of the `Object` class has been overridden with the return type as `String`. This method is basically used for returning a `String` that identifies an object. This method is automatically invoked when we try to print an object. It can also be explicitly invoked with the help of an object.

L3 Returns a string `"My Demo Object created"`.

L5 Within the print statement, an object of class `Demo` is created and whenever an attempt to print an object occurs, the `toString()` method is called automatically.

Note You may now rewrite the complex number program to add the `toString()` method to it instead of `display` method of that class.

6.3.2 Java Wrapper Classes

Java primitive types are not objects, i.e., we cannot term Java as a pure object-oriented language. The language designers decided that the higher processing speed and memory efficiency of simple, non-class structures for such heavily used data types simply overweighed the elegance of a pure object-only language.

For each primitive type, there is a corresponding `wrapper` class designed. As the name suggests, the `wrapper` class is a wrapper around a primitive data type. These classes represent primitive data types, e.g., a `boolean` data type can be represented as a `Boolean` class instance.

As we have said earlier, an instance of a wrapper contains or *wraps* a primitive value of the corresponding type. Wrappers allow for situations where primitives cannot be used but their corresponding objects are required. For example, a very useful tool is the `ArrayList` class (see Chapter 10), which is a list that can grow or shrink, unlike an array. So if one wants to use an `ArrayList` to hold a list of numbers, the numbers must be wrapped in an integer instance. Mostly you will use wrapper class methods to convert a numeric value to a string or vice versa.

Table 6.4 Wrapper for Primitive Types

Primitive	Wrapper
`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`char`	`java.lang.Character`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`
`void`	`java.lang.Void`

Table 6.4 lists the primitive data types and their corresponding wrapper classes.

You can easily make out that except for integer; the wrappers come with the same name as the corresponding primitive type except that the first letter is capitalized. Wrappers are normal classes that extend the `Object` as a superclass like all Java classes. The wrapper constructors create class objects from the primitive types. For example, for a double floating point number "d":

```
double a = 4.3; Double wrp = new Double(a);
```

Here a `Double` wrapper object is created by passing the double value in the `Double` constructor argument. In turn, each wrapper provides a method to return the primitive value.

```
double r = wrp.doubleValue();
```

Each wrapper has a similar method to access the primitive value: `intValue()` for integer, `booleanValue()` for boolean, and so on.

Features of Wrapper Classes Some of the sound features maintained by the wrapper classes are as under:

- All the wrapper classes except `Character` and `Float` have two constructors—one that takes the primitive value and another that takes the String representation of the value. `Character` has one constructor and `float` has three.
- Just like strings, wrapper objects are also immutable, i.e., once a value is assigned it cannot be changed.

Wrapper Classes: Constructors and Methods

The wrapper classes have a number of static methods for handling and manipulating primitive data types and objects. The methods along with their usage are listed below:

Constructors Converting primitive types to wrapper objects.

```
Integer ValueOfInt = new Integer(v)    // primitive integer to integer object
Float ValueOfFloat = new Float(x)      // primitive float to float object
Double ValueOfDouble = new Double(y)   // primitive double to double object
Long ValueOfLong = new Long(z)         // primitive long to long object
```

Here v, x, y, and z are int, float, double, and long values, respectively. There is one more way of converting a primitive value to a wrapper, the valueOf() method, which we will discuss later.

Ordinary Methods

Converting Wrapper Objects to Primitives All the numeric wrapper classes have six non-static methods, which can be used to convert a numeric wrapper to their respective primitive numeric type. These methods are byteValue(), doubleValue(), floatValue(), intValue(), longValue(), and shortValue(). Some of them are used as follows:

```
int v = ValueOfInt.intValue();          // Converting wrapper object to primitive integer
float x = ValueOfFloat.floatValue();    // Converting wrapper object to primitive float
long y = ValueOfLong.longValue();       // Converting wrapper object to primitive long
double z = ValueOfDouble.doubleValue(); // Converting wrapper object to primitive double
```

Converting Primitives to String Object The method toString() is used to convert primitive number data types to String, as shown below:

```
String xyz = Integer.toString()         // Converting primitive integer to String
String xyz = Float.toString()           // Converting primitive float to String
String xyz = Double.toString()          // Converting primitive double to String
String xyz = Long.toString()            // Converting primitive long to String
```

Parser Methods

Converting Back from String Object to Primitives The six parser methods are parseInt, parseDouble, parseFloat, parseLong, parseByte, and parseShort. They take a string as the argument and convert it to the corresponding primitive. They throw a NumberFormatException if the value of the String does not represent a proper number. Parser methods can be used as shown below:

```
int v = Integer.parseInt(xyz)
// For converting String containing int values like "10" to primitive integer

long y = Long.parseLong(xyz)
// For converting String containing long values like "123456" to primitive integer
```

Converting Primitive Value Represented by String Object to Wrapper Object All wrapper classes define a static method called valueOf(), which returns the wrapper object corresponding to the primitive value represented by the string argument as shown below. valueOf() is overloaded: one version accepts integer values and another accepts a String. String argument method generates a NumberFormatException in case the value in a String does not contain a number.

```
Double ValueOfDouble = Double.valueOf(xyz);
// For converting String containing double values to wrapper objects

Float ValueOfFloat = Float.valueOf(xyz);
// For converting String containing float values to wrapper objects

Integer ValueOfInteger = Integer.valueOf(xyz);
// For converting String containing int values to wrapper objects
```

```
Long ValueOfLong = Long.valueOf(xyz);
// For converting String containing long values to wrapper objects

Double ValueOfDouble = Double.valueOf(xyz);
// For converting primitive value double to wrapper objects

Float ValueOfFloat = Float.valueOf(xyz);
// For converting primitive values float to wrapper objects

Integer ValueOfInteger = Integer.valueOf(xyz);
// For converting int to wrapper objects

Long ValueOfLong = Long.valueOf(xyz);
// For converting long to wrapper objects
```

Binary and Hexadecimal Conversion The following method converts an integer to its binary/hexadecimal equivalent and returns it as a String object.

```
System.out.println(Integer.toBinaryString(8));
System.out.println(Integer.toHexString(32));
```

The integer value 8 is converted to its binary equivalent using `toBinaryString()`, i.e., 1000, and 32 is converted to its hexadecimal equivalent, i.e. 20.

Autoboxing and Unboxing of Wrappers

Java 5.0 introduced a new feature for converting back and forth between a wrapper and its corresponding primitive. The conversion from primitives to wrappers is known as *boxing*, while the reverse is known as *unboxing*.

In the previous section, we have already seen boxing and unboxing being enforced by the use of a certain amount of clumsy code. Before J2SE 1.5, Java had primitive data types with wrappers around them, so programmers had to convert from one type to another programmatically.

```
public void manualConversion()
{
  int a = 12;
  Integer b = Integer.valueOf(a);
  int c = b.intValue();
}
```

If you are dealing with a lot of instances of wrappers and conversions, you will need to deal with a lot of method invocations. The to and fro conversion between primitives and wrappers is simplified by the use of *autoboxing* and *unboxing*. Behind the scenes, the compiler creates codes to implicitly create objects for you.

```
public void autoBoxing()
{
  int a = 12;
  Integer b = a; // wrapping
  int c = b;
}
```

Here, the wrapping is done automatically. There is no need to explicitly call the integer constructor. Autoboxing means a primitive value is automatically converted into the wrapper object. The reverse process, i.e., automatic conversion back from wrapper object to primitive value, is known as *unboxing*.

To sum up the complete essence of autoboxing and unboxing, we take the following piece of code:

```
Integer wrap_int = 5;          //primitive 5 autoboxed into an Integer object
int prim_int = wrap_int;       //automatic unboxing of Integer into int
```

There is one thing that you must remember: boxing and unboxing too many values can put undue pressure on the garbage collector.

6.3.3 String Class

Strings are basically immutable objects in Java. Immutable means once created, the strings cannot be changed. In fact there is a class named `String` in the `java.lang` package for creating strings. Whenever we create strings, it is this class that is instantiated. In Java, strings can be instantiated in two ways:

```
L1    String x = "String Literal Object";
L2    String y = new String ("String object is created here");
```

L1 Shows a string literal being assigned to a string reference: `x`.

L2 Shows the creation of a string object with the help of `new` keyword and the string literal is passed as an argument to the constructor. Does it mean that in L1, no object is created? Well actually an object of class `String` is created in both the lines, the only difference is that in L1, it is created implicitly and the memory is allocated from a memory pool which is created specifically for string literals. In L2, the object is created explicitly using the `new` keyword, so the memory required for the object is allocated out of the memory pool.

Before creating objects for string literals (L1), JVM checks the memory pool for the existence of string literals in the pool and if found, a reference to the existing String object is passed, else a new string instance in the pool is created and it is returned. In other words, string objects in the pool are shared and because it is a sharable thing, it is made immutable so that strings may not become inconsistent and corrupt. The concept of memory pool for string literals was created to save time (speed up working) and memory because strings are very often used by programmers. Let us take an example to clearly understand the concept.

Example 6.8 String Creation and Test for Equality

```
      class StringTest
      {
      public static void main(String args[]){
L1     String a = "Hello";
L2     String b = "Hello";
L3     String c = new String("Hello");
L4     String d = new String("Hello");
L5     String e = new String("Hello, how are you?");
L6     if(a == b)
```

```
L7     System.out.println("object is same and is being shared by a & b");
       else
L8     System.out.println("Different objects");
L9     if(a == c)
L10          System.out.println("object is same and is being shared by a & c");
else
L11          System.out.println("Different objects");
L12  if(c == d)
L13          System.out.println("same object");
       else
L14          System.out.println("Different objects");
L15          String f = e.intern();
L16  if(f == a)
L17          System.out.println("Interned object f refer to the already created object a
             in the pool");
       else
L18          System.out.println("Interned object does not refer to the already created
             objects, as literal was not present in the pool. It is a new object which has
             been created in the pool");
       }}
```

Output

```
C:\examples\chap 6>java StringTest
object is same and is being shared by a & b
Different objects
Different objects
Interned object does not refer to the already created objects, as literal was not present
in the pool. It is a new object which has been created in the pool.
```

Explanation

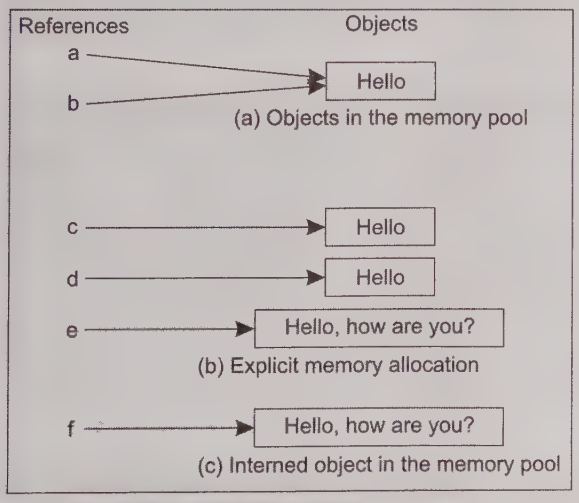


Fig. 6.3 String Objects

L1 An implicit string object has been created for a string literal `Hello` in the memory pool and the reference to the newly created object in the pool is return to `a`.

L2 As explained earlier, string literal is same as that of L1, i.e., `Hello`. JVM does not create a new object but passes the reference of the previously created object (L1) to `b`, which means `a` and `b` now point to the same object. See the following Fig. 6.3.

L3 Although the same string literal `Hello` is being used, but the object created is not a part of the memory pool, as the keyword `new` is used for object creation. Whenever `new` is used to create an object, it is allocated explicit memory and that memory is apart from the memory pool of strings.

L4 A new object is created which is different from `a`, `b`, and `c` as well.

L5 A new string object is created with a different literal this time `"Hello, how are you?"`. This object is also different from all the objects that we have created till this point in our example.

L6 Equality operator (= =) is used in the `if` statement to check whether both references `a` and `b` are pointing to the same location or not. Equality operator is not used for matching the contents of the strings, i.e., literals. The two references that are being matched are `a` and `b` and as both point to the same location, L7 is executed. If `a` and `b` do not point to the same location, L8 would be executed.

L7 Print statement to show that references point to the same object.

L8 Print statement to show that references point to different objects.

L15 As already discussed, `Strings` created with the help of `new` are not allocated memory from the pool, but are interned. The method `java.lang.String.intern()` is used for this purpose. The `intern()` method creates a string object in the pool with the same `String` literal as that of the invoking `String` object and returns a reference of the newly created object in the pool. In this Line, `f` points to the newly created object in the pool because the string literal object `Hello, how are you?` does not exist in the pool. The `intern` method is called from the string object which needs to be interned, i.e., the previous String object will be garbage collected as it is no longer in use.

L16 Checks whether `f` and `e` point to the same object in the pool or not. They actually point to different locations and that is why L18 gets executed.

String Manipulation

`Strings` in Java are immutable (read only) in nature. That is, once the `Strings` are defined, they cannot be altered. Let us have a look at the following lines of code:

```
L1    String x = "Hello";          // ok
L2    String x = x +"World";       // ok, but how?
```

Java does not support operator overloading, but the '+' operator is already overloaded to accept different operands and it acts accordingly. If at least one of the operand is a string, it concatenates. The question that arises is that if strings are immutable, then how L2 gets executed? Actually L2 gets converted into the following statement:

```
String x = new StringBuffer().append(x).append("World").toString();
```

A new `StringBuffer` object is created which is used for the mutable set of characters. Mutable characters can change their values. The `append` (add at the end) method of the `StringBuffer` object is used to append the string `Hello` contained in `x` into the newly created `StringBuffer` object. Again the `append` method is used to append the string `World` to existing `Hello` in the new object. The method `toString()` converts `StringBuffer` object back to `String` and `x` points to this newly created `String` object. No references exist for the existing object `Hello`; it will be garbage collected.

String Methods

The `String` class provides a lot of methods. Table 6.5 lists a few common methods of the `String` class.

Table 6.5 Few Methods of `string` Class

Method Name with Signature	Method Details
`int length()`	To find the length of the string.
`boolean equals(String str)`	Used to check the equality of String objects. In contrast to == operator, the check is performed character by character. If all the characters in both the Strings are same, it returns true, else false.
`int compareTo(String s)`	Used to find whether the invoking String (Fig. 6.2) is Greater than, less than or equal to the String argument. It returns an integer value. If the integer value is (a) less than zero – invoking string is less than String argument (b) greater than zero – invoking String is greater than String argument (c) equal to zero – invoking String and String argument are equal
`boolean regionMatches (int startingIndx, String str, int strStartingIndx, int numChars)`	Matches a specific region of the String with a specific region of the invoking String. The argument details: **startingIndx**—specifies the region from the invoking String to be matched. **str**—is the second string to be matched. **strStartingIndx**—specifes the region from the string to be matched with the invoking String. **numChars**—specifies the number of characters to be matched in both strings from their respective starting indexes.
`int indexOf(char c)`	To find the index of a character in the invoking `String` object.
`int indexOf(String s)`	Overloaded method to find the starting index of a String argument in the invoking `String` object.
`int lastIndexOf(char c)`	To find the last occurrence of a character in the invoking String.
`int lastIndexOf(String s)`	Overloaded method to find the last occurrence of the String argument in the invoking `String` object.
`String substring(int s Index)`	To extract the `String` from the invoking `String` object starting with Index till the End of the `String`.
`String substring(int startingIndex, int endingIndex)`	Overloaded method to extract the String starting with starting Index till the ending Index from the invoking String object string.
`int charAt(int pos)`	To find the character at a particular position (pos).
`String toUpperCase()`	To change the case of an entire string to capital letters.
`String toLowerCase()`	To change the case of an entire string to small letters.
`boolean startsWith(String ss)`	To find whether an invoking string starts with a string argument.
`boolean endsWith(String es)`	To find whether an invoking string ends with a string argument.
`Static String valueOf(int is)`	Converts primitive type `int` value to string.
`Static String valueOf(float f)`	Overloaded static method to convert primitive type float value to string.
`Static String valueOf(long l)`	Overloaded static method to convert primitive type long value to string.
`Static String valueOf(double d)`	Overloaded static method to convert primitive type double value to string.

Example 6.9 String Class Methods

```
class StringDemo {
public static void main(String args[]){
// String Declaration
String x = "This is a Demo String";
String y = "This is a Demo String 2";

// int declaration
int i = 20;
// finding the length of String
L1  System.out.println("Length of String = " +x.length());
/* equals method of Object class has been overridden by the String class for per-
forming different function i.e., equating two string objects by matching strings
character by character */
L2  System.out.println("x and y are equal = " +(x.equals(y)));
// comparison of Strings
L3  if((x.compareTo(y)) < 0)
L4       System.out.println("x is less than y");
L5  else if((x.compareTo(y)) > 0)
L6       System.out.println("x is greater than y");
L7  else
L8       System.out.println("x is equal to y");
         // Region Matching within Strings
L9       System.out.println("x region matches with y:" + ((x.regionMatches(0,y,0,11)));
         // finding index of Characters
L10      System.out.println("index of \" i\" in String x is: " +x.indexOf("i"));
         // finding index of particular String
L11      System.out.println("index of \"is\" in String x is: " +x.indexOf("is"));

         // finding the last occurrence of a particular character
L12      System.out.println("Last index of \"i\" in String x is: " +x.lastIndexOf("i"));
         // finding the last occurrence of a particular character
L13      System.out.println("Last index of \"is\" in String x is: " +x.lastIndexOf("is"));
         // sub string
L14      System.out.println("Substring of String x from character 4 is: " +x.substring(4));
L15      System.out.println("Substring of String x from character 4 to 15 is:
         " +x.substring(4,15));

         // finding character at particular position
L16      System.out.println("character at position 6 is:" +x.charAt(6));

         // upper case and lower case
L17      System.out.println("UpperCase: " +x.toUpperCase());
L18      System.out.println("LowerCase: " +x.toLowerCase());

         // finding whether strings start and end with a particular string
L19      System.out.println("x starts with \"Th\":" +x.startsWith("Th"));
```

```
L20         System.out.println("x ends with \" Th\": " +x.endsWith("Th"));
L21         System.out.println("Converts int to String: " +String.valueOf(i));
            }}
```

Output

```
C:\javabook\ chap 6>java StringDemo
Length of String = 21
x and y are equal = false
x is less than y
x region matches with y : true
index of "i" in String x is: 2
index of "is" in String x is: 2
Last index of "i" in String x is: 18
Last index of "is" in String x is: 5
Substring of String x from character 4 is: is a Demo String
Substring of String x from character 4 to 15 is: is a Demo
character at position 6 is: s
UpperCase: THIS IS A DEMO STRING
LowerCase: this is a demo string
x starts with "Th" : true
x ends with "Th" : false
converts int to String: 20
```

Explanation

As discussed earlier, all instance methods of the `String` class are invoked with the help of String objects and class methods through the `String` class name. Table 6.4 describes these functions in brief. Figure 6.4 pictorially depicts how are the methods of `String` class invoked.

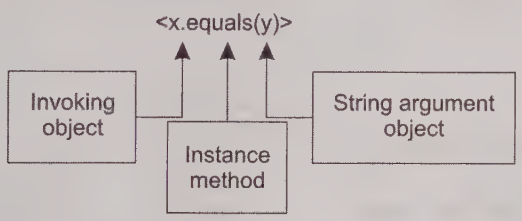


Fig. 6.4 Invoking the Methods of `String` Class

6.3.4 `StringBuffer` Class

The `StringBuffer` class is used for representing changing strings. As already discussed, `StringBuffer` offers more performance enhancement whenever we change strings, because it is this class that is used behind the curtain. So it is advisable to use `StringBuffer` rather than `String` in such a situation. If `String` class is used, it would result in wastage of memory and time, as temporary string objects would be needed while changing strings. `StringBuffer` contains a sequence of characters which can be altered through the methods of this class. Just like any other buffer, `StringBuffer` also has a capacity and if the capacity is exceeded, then it is automatically made larger. The initial capacity of `StringBuffer` can be known by using a method `capacity()`. A few common methods of the `StringBuffer` class are shown in Table 6.6.

Table 6.6 Methods of `StringBuffer` Class

Method name with signature	Method details
`int capacity()`	Returns the current capacity of the storage available for characters in the buffer. When the capacity is approached, the capacity is automatically increased.
`StringBuffer append(String str)`	Appends String argument to the buffer
`StringBuffer replace (int sindx,int eIndx,String str)`	The characters from start to end are removed and the string is inserted at that position
`StringBuffer reverse()`	Reverses the buffer character by character
`Char charAt(int index)`	Returns the character at the specified index
`Void setCharAt(int indx,char c)`	Sets the specified character at the specified index

Example 6.9 shows how the `StringBuffer` class is used in a program.

Example 6.10 `StringBuffer` Object

```
    class StringBufferDemo {
    public static void main(String args[]){
L1    StringBuffer sb = new StringBuffer();
L2    System.out.println("Initial Capacity : " +sb.capacity());
L3    System.out.println("String appended : " +sb.append ("Dogs bark at night"));
L4    System.out.println("String replaced: " +sb.replace  (10,12,"during"));
L5    System.out.println("String reversed : " +sb.reverse());
L6    System.out.println("Current Capacity : " +sb.capacity());
L7    System.out.println("character at position 3 is: " + sb.charAt(3));
L8    sb.setCharAt(3,'a');
L9    System.out.println("sb after setting \"a\" at 3: " +sb);
    }}
```

Output

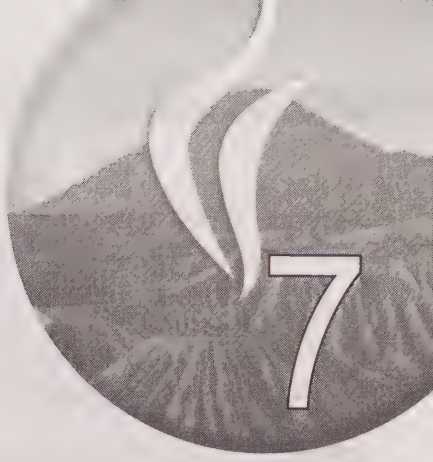
```
C:\javabook>java StringBufferDemo
Initial Capacity : 16
String appended : Dogs bark at night
String replaced : Dogs bark during night
String reversed : thgin gnirud krab sgoD
Current Capacity : 34
character at position 3 is: i
sb after setting "a" at 3: thgan gnirud krab sgoD
```

Explanation

The description of the methods used in the program is available in Table 6.6.

6.3.5 `StringBuilder` Class

Java 5 introduced a substitute of `StringBuffer`: the `StringBuilder` class. This class is faster than `StringBuffer`, as it is not synchronized. The methods of both the classes are same with the exception that the methods (`append()`, `insert()`, `delete()`, `deleteCharAt()`, `replace()`, and



Exception, Assertions, and Logging

When the imagination and willpower are in conflict, are antagonistic, it is always the imagination which wins, without any exception. **Emile Coue**

After reading this chapter, the readers will be able to

- understand the concepts and applications of exception handling
- understand all the keywords used for exception handling
- create user-defined exceptions
- know what assertions are and how to use them
- know the basics of logging

7.1 INTRODUCTION

Exceptions in real life are rare and are usually used to denote something unusual that does not conform to the standard rules. For example, Abraham Lincoln was an exception who, despite all hurdles in his life, rose to become the sixteenth president of the USA. In computer programming, exceptions are events that arise due to the occurrence of unexpected behavior in certain statements, disrupting the normal execution of a program.

Exceptions can arise due to a number of situations. For example,

- Trying to access the 11th element of an array when the array contains only 10 elements (`ArrayIndexOutOfBoundsException`)
- Division by zero (`ArithmeticException`)
- Accessing a file which is not present (`FileNotFoundException`)
- Failure of I/O operations (`IOException`)
- Illegal usage of null (`NullPointerException`)

There are predefined classes (mentioned in the parenthesis above) for all exception types representing each such situation. The topmost class in the hierarchy is `java.lang.Throwable`. This class has two siblings: `Error` and `Exception`. All the classes representing exceptional conditions are subclasses of the `Exception` class. Whenever an exception occurs in a method, the runtime environment identifies the type of `Exception` and throws the object of it. If the method does not

Table 7.1 Checked and Unchecked Exception Classes

Checked Exceptions	Unchecked Exceptions
`ClassNotFoundException`	`ArithmeticException`
`NoSuchFieldException`	`ArrayIndexOutOfBoundsException`
`NoSuchMethodException`	`NullPointerException`
`InterruptedException`	`ClassCastException`
`IOException`	`BufferOverflowException`
`IllegalAccessException`	`BufferUnderflowException`

Figure 7.2 shows the exception hierarchy in Java. Not all the `Exception` and `Error` subclasses have been depicted in the figure. The dots in the diagram are an indicator that there are other classes also within the immediate superclass. Example 7.1 can be modified to handle the exception generated in `method3`.

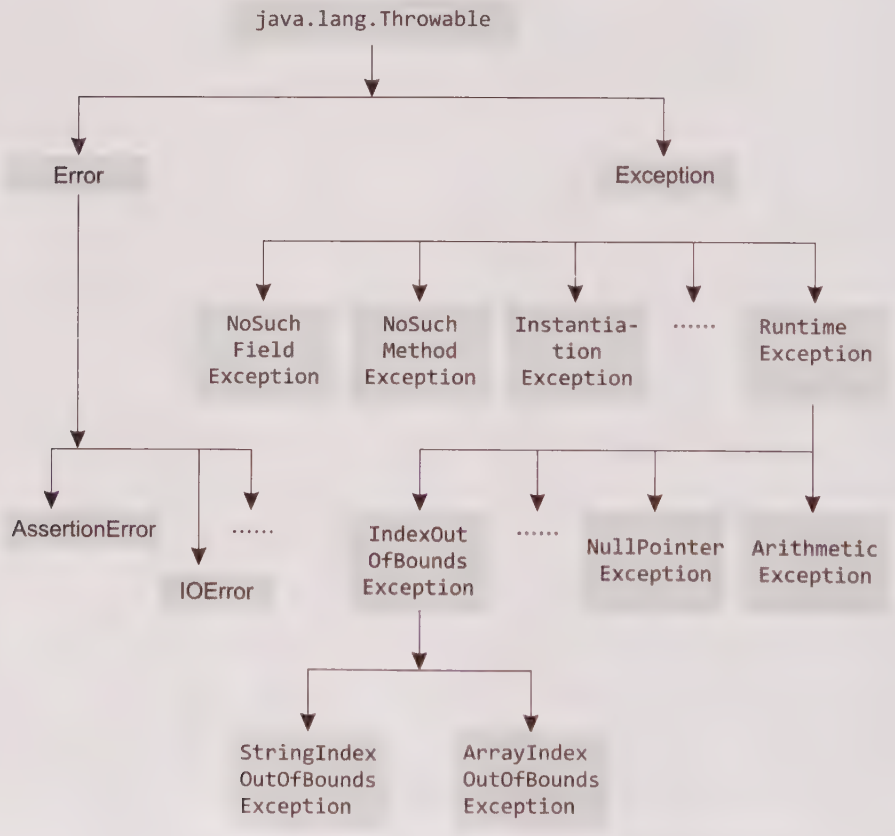


Fig. 7.2 Exception Hierarchy

7.2 EXCEPTION HANDLING TECHNIQUES

Java provides five keywords for exception handling: `try`, `catch`, `throw`, `throws`, and `finally`. Let us take a look at all these one by one.

7.2.1 try...catch

The `try/catch` block can be placed within any method that you feel can throw exceptions. All the statements to be tried for exceptions are put in a `try` block and immediately following the `try` is the `catch` block. `catch` block is used to catch any exception raised from the `try` block. If exception occurs in any statement in the `try` block, the following statements are not executed and control immediately passes to the corresponding `catch` block.

Example 7.2 **try...catch**

```
L1   class ExDemo1
L2   {
L3   public static void main(String args[])
L4   {
L5        method1();
L6   }
L7   static void method1()
L8   {
L9        System.out.println("IN Method 1, Calling Method 2");
L10       method2();
L11       System.out.println("Returned from method 2");
L12  }
L13  static void method2()
L14  {
L15  System.out.println("IN Method 2, Calling Method 3");
L16  try{
L17       method3(); }
L18  catch(Exception e)
L19  {
L20       System.out.println("Exception Handled");
L21  }
L22  System.out.println("Returned from method 3");
L23  }
L24  static void method3()
L25  {
L26       System.out.println("IN Method 3");
L27       int a = 20,b = 0;
L28       int c = a/b;
L29       System.out.println("Method 3 exits");
     }}
```

Output

```
C:\javabook\programs\chap 7>java ExDemo1
IN Method 1, Calling Method 2
IN Method 2, Calling Method 3
IN Method 3
Exception Handled
Returned from method 3
Returned from method 2
```

Explanation

L28 Exception occurred. No handling mechanism in `method3()`, so the control passes to the `try...catch` block in `method2()`.

L29 It is not executed, as the statements following the occurrence of an exception are not executed.

L16 `try` block declared. The statements to be monitored for exceptions should be placed in the `try` block within a method.

L17 A call to `method3()` is placed within the `try` block.

L18 `catch` clause defined with an argument of type `Exception` (parent class) so that the exception objects thrown from the `try` block can be caught here. *A superclass reference variable can refer to a subclass object.* The `try` block is immediately followed by a `catch` block. As soon as an exception is encountered in the `try` block, statements following the statement on which the exception occurred are not executed. The runtime environment creates an object of class representing the exception and throws it. Control passes to the appropriate `catch` block (first appropriate `catch` in case multiple `catch` clauses are present) where the thrown object is caught and assigned to `e`, i.e., the `Exception` reference variable.

L20 Prints `ExceptionHandled`.

L22 Prints `Returned from method3()`. After the exception has been caught (`try...catch` mechanism implemented), execution resumes as normal. This was not possible in Example 7.1. After this, the control passes back to `method1()` from where `method2()` was called and L11 gets executed (see output).

A single `try` can have multiple `catch` clauses, for catching specific exceptions. As soon as an exception is thrown, the first appropriate `catch` clause responsible for handling that exception is located and the exception is passed to it. By first appropriate catch, we mean, if `ArrayIndexOutOfBoundsException` is generated, then the control passes to the first catch that either specifies the `ArrayIndexOutOfBoundsException` or the `IndexOutOfBoundsException` superclass of the `ArrayIndexOutOfBoundsException` or `Exception`. All exceptions can be caught by the `Exception` class. Example 7.3 shows how multiple catch clauses are incorporated in a program.

Example 7.3 Multiple Catch Clauses

```
L1   class Multiple_Catch
L2   {
L3   public static void main(String args[])
L4   {
L5     method1();
L6   }
L7   static void method1()
L8   {
L9     System.out.println("IN Method 1, Calling Method 2");
L10    method2();
L11    System.out.println("Returned from method 2");
L12  }
L13    static void method2()
L14    {
L15         System.out.println("IN Method 2, Calling Method 3");
L16    try {
L17    method3(); }
L18    catch(ArithmeticException ae)
```

```
L19     {
L20          System.out.println ("Arithmetic Exception Handled: " +ae);
L21     }
L22     catch(Exception e)
L23     {
L24          System.out.println("Exception Handled");
L25     }

L26          System.out.println("Returned from method 3");
L27     }
L28     static void method3()
L29     {
L30          System.out.println("IN Method 3");
L31          int a = 20, b = 0;
L32          int c = a/b;
L33          System.out.println("Method 3 exits");
        }}
```

Output

```
C:\javabook\programs\chap 7>java Multiple_Catch
IN Method 1, Calling Method 2
IN Method 2, Calling Method 3
IN Method 3
Arithmetic Exception Handled: java.lang.ArithmeticException: / by zero
Returned from method 3
Returned from method 2
```

Explanation

L16 `try` block defined.

L17 Call to `method3()`.

L18 The first `catch` clause defined with an argument of type `ArithmeticException` class.

L20 Prints `Exceptionhandled` concatenated with the output of `ae.toString()`. Remember `toString()` is called automatically when you try to print any object. `toString()` method is overridden in the `ArithmeticException` class to print its own `string` rather than that of the `Object` class.

L22 The second `catch` clause defined with an argument of type `Exception` class. This has been specified intentionally because if any other exception is thrown apart from `ArithmeticException`, then that exception will be caught in this particular `catch` clause.

Note While specifying multiple catch clauses for exception handling, the `catch` clause having the `Exception` type as its argument should be the last catch block in your program. This is because if the `catch` having the reference variable of type `Exception` class is placed as the top catch clause, then all the exceptions thrown from the try block will be caught in the first catch and the control will never pass onto the lower catch blocks, leading to an unreachable code.

An unreachable code in Java is easily recognized by the Java compiler and it complains about it during compilation. For example, if the `catch` clauses in Example 7.3 are reversed, as shown, the program will not compile.

```
catch(Exception e){}
catch(ArithmeticExceptionae) {}
```

7.2.2 throw Keyword

The `throw` keyword is used to explicitly throw an exception. In the earlier examples, this job was being done implicitly. Whether implicit or explicit, objects of exception need to be created before they are thrown. Execution of the program is suspended as in previous cases and the runtime environment looks for the appropriate catch to handle the exception. `throw` is more useful when we want to throw a user-defined exception. The syntax for `throw` is as follows:

```
throw new NullPointerException(); // throw new ThrowableInstance
```

Let us rework Example 7.3 to throw an exception explicitly.

Example 7.4 throw Keyword

```
L1   class ThrowDemo
L2   {
L3     public static void main(String args[])
L4     {
L5          method1();
L6     }
L7     static void method1()
L8     {
L9          System.out.println("IN Method 1, Calling Method 2");
L10         method2();
L11         System.out.println("Returned from method 2");
L12    }
L13    static void method2()
L14    {
L15         System.out.println("IN Method 2, Calling Method 3");
L16         try {
L17         method3(); }
L18         catch(Exception e)
L19         {
L20                 System.out.println("Exception Handled:" + e);
L21         }
L22         System.out.println("Returned from method 3");
L23    }
L24    static void method3()
L25    {
L26         System.out.println("IN Method 3");
L28         throw new ArithmeticException("Testing Throw");
       // This line is intentionally commented. If not, it results
       // in compile time error as it leads to unreachable code.
L29    // System.out.println("Method 3 exits");
     }}
```

Output

```
C:\javabook\programs\chap 7>java ThrowDemo
IN Method 1, Calling Method 2
IN Method 2, Calling Method 3
```

```
IN Method 3
Exception Handled: java.lang.ArithmeticException: Testing throw
Returned from method 3
Returned from method 2
```

Explanation

L28 Instead of an expression which leads to the runtime environment throwing an exception, we have used `throw` keyword to throw the exceptions ourselves. Just like the runtime environment, we also need to create an object for throwing it. So the `ArithmeticException` object is created with the help of `new` keyword and an argument is passed to its constructor. This argument is printed onto the console via the `toString()` method of the `ArithmeticException` class, when we catch this exception and print the exception object (see output). This argument can also be separately printed using the `getMessage()` method of the `ArithmeticException` class.

L29 It is commented. If it is not commented, the compiler will give an error stating Unreachable Code. Particularly, in this program, the control will always move out after the throws clause, searching for a handler, so this line will never be executed. The Java compiler is intelligent enough to understand this and raises an error.

This exception is caught and printed in the catch present in L18–21. Rest of the logic is similar to the previous example.

7.2.3 throws

The `throws` is added to the method signature to let the caller know about what exceptions the called method can throw. It is the responsibility of the caller to either handle the exception (using `try…catch` mechanism) or it can also pass the exception (by specifying `throws` clause in its method declaration). If all the methods in a program pass the exception to their callers (including `main()`), then ultimately the exception passes to the default exception handler. A method should use either of the two techniques—`try/catch` or `throws`. Usually (for checked exceptions specifically), it is the `catch` or `specify` mechanism that is used. A method can throw more than one exception; the exception list is specified as separated by commas. The syntax for the `throws` keyword is shown below:

```
public void divide(int a, int b) throws ArithmeticException, IllegalArgumentException
```

Let us take a look at the following example.

Example 7.5 **throws Keyword**

```
L1  class ThrowsDemo
L2  {
L3    public static void main(String args[])
L4    {
L5         method1();
L6    }
L7    static void method1()
L8    {
L9         System.out.println("IN Method 1, Calling Method 2");
L10        method2();
L11        System.out.println("Returned from method 2");
```

```
L12     }
L13     static void method2()
L14     {
L15     System.out.println("IN Method 2, Calling Method 3");
L16     try{
L17          method3(4,0); }
L18     catch(Exception e)
L19          {
L20                  System.out.println("Exception Handled: " + e);
L21          }
L22          System.out.println("Returned from method 3");
L23     }
L24     static void method3(int a, int b) throws Exception
L25     {
L26     System.out.println("IN Method 3");
L27     if(b == 0)
L28          throw new ArithmeticException("Testing throw");
L29     else
             System.out.println("Result: "+a/b);
        }}
```

Output

When a = 4 and b = 2

```
C:\javabook\programs\chap 7>java ThrowsDemo
IN Method 1, Calling Method 2
IN Method 2, Calling Method 3
IN Method 3
Result: 2
Returned from method 3
Returned from method 2
```

When a = 4 and b = 0

```
C:\javabook\programs\chap 7>java ThrowsDemo
IN Method 1, Calling Method 2
IN Method 2, Calling Method 3
IN Method 3
Exception Handled: java.lang.ArithmeticException: Testing throw
Returned from method 3
Returned from method 2
```

Explanation

L24 `method3()` has been declared with `throws` clause specifying that it may throw an exception. The parent class (Exception) has been specified in the `throws` clause, so there is no need to explicitly mention the subclass name (`ArithmeticException`). If `throws` is omitted in this line, the program works as usual. If the `try/catch` in `method2()` (L16, L18–21 and L23) is omitted and `throws` in `method3()` declaration is kept intact, the compiler will not compile the program as now, it is mandatory for the calling method to either pass or catch the exception.

L26 Print statement.

L27 `if` statement checks the value of `b`. If it is zero, L28 is executed, else L29.

L28 An object of `ArithmeticException` is created and thrown.

L29 `else` prints the result of division of `a` by `b`.

7.2.4 finally Block

The `finally` block is always executed in `try-catch-finally` statements irrespective of whether an exception is thrown from within the `try/catch` block or not. Statements following the exception in a `try` block are not executed. Some statements are mandatory to execute such as the statements related to the release of resources. All these statements can be put in a `finally` block. The syntax of the `finally` keyword is as follows:

```
try {...} catch(Throwable e){...} finally {....}
```

Let us take an example to understand it better.

Example 7.6 finally Keyword

```
L1  class FinallyDemo
L2  {
L3    public static void main(String args[])
L4    {
L5         method1();
L6         System.out.println("Result : "+method2 (24,0)); }
L7         static void method1()
L8         {
L9         try {
L10               System.out.println("IN Method 1");
L11               throw new NullPointerException(); }
L12        catch(Exception e)
L13        {
L14               System.out.println("Exception Handled: " + e);
L15        }
L16        finally {
L17        System.out.println("In method 1 finally"); } }
L18        static int method2(int a, int b)
L19        {
L20        try{
L21               System.out.println("IN Method 2");
L22               return a/b; }
L23        finally {
                  System.out.println("In method 2 finally");
           }
      }
   }
```

Output

When a = 24 and b = 4

```
C:\javabook\programs\chap 7>java FinallyDemo
IN Method 1
Exception Handled: java.lang.NullPointerException
In method 1 finally
IN Method 2
In method 2 finally
Result : 6
```

When a = 24 and b = 0

```
C:\javabook\programs\chap 7>java FinallyDemo
IN Method 1
Exception Handled: java.lang.NullPointerException
In method 1 finally
IN Method 2
In method 2 finally
Exception in thread "main" java.lang.ArithmeticException: / by zero
at FinallyDemo.method2(FinallyDemo.java:24)
at FinallyDemo.main(FinallyDemo.java:6)
```

Explanation

L5 Call to `method1()`. Control passes to L7.

L6 Call to `method2()` and if any, return is printed on the screen.

L7 `method1()` declaration.

L9 `try` block defined.

L11 `NullPointerException` is thrown. Control passes to catch in L12.

L12 `catch` block corresponding to try in L9.

L14 Prints the exception object `e`. (`e.toString()` is called by default).

L16 Shows the `finally` block. The exception thrown in L11 is caught at L12. The `finally` block following catch gets executed after that (see output).

L17 The statement within `finally` gets executed.

L18 `method2()` declared expecting two integer arguments. Value passes are 24 and 0.

L20 `try` block within `method2()`.

L22 As already discussed, the value of `b` being zero, an attempt to divide any number by zero results in an `ArithmeticException` being thrown.

L23 Just to show that the `finally` block executes in all cases, we have intentionally not given the catch in `method2()`. A `try` can either have a corresponding `catch` with `finally` or it can also have a `finally` following it. In the earlier examples, we have seen that as soon as an exception is encountered, its appropriate handler is looked upon and nothing gets executed until and unless the exception is handled. The only exception to this fact is the `finally` block. In our example, the exception is thrown in L22. `method2()` does not have its own catch to handle exceptions, so its caller is to be looked upon but before control passed to the caller, i.e., main method, the `finally` in `method2()` is executed. And then the control passes to `main()` where no handler is present, so the runtime environment handles the exception as already discussed (see output).

7.2.5 try-with-resources Statement

Java 7 added a new enhancement to the exception handling mechanism, i.e., automatic resource management with a `try-with-resources` statement. The applications uses many resources during their lifetime by creating their objects, e.g., creating a data base connection for accessing/updating databases, or creating file objects for working with files, or creating sockets for transmission/receiving of data, etc. A common mistake committed by programmers is that they often do not close/release the resources occupied by the programs, after their task is complete. This leads to many orphaned instances, inefficient memory allocation, and garbage collection. Hence the need for automatic resource management arises.

To address this problem `AutoCloseable`, a new interface has been created in the `java.lang` package. The resources that want to be closed must implement this interface. This interface has just one method,

```
public void close() throws Exception
```

This close method will be overridden by the class that implements the interface and all resources releasing code can be put in this method. The close method of the `AutoCloseable` object is called automatically when it is used with a `try-with-resources` statement as soon as the `try-with-resources` block has finished execution regardless of whether an exception is thrown or not. The syntax of a `try-with-resources` statement is as follows:

```
try (resources to be used and automatically released)
{
     // statements within the block
}
```

For example

```
try (abc a=new abc(); pqr p=new pqr())
{
     // statements within the block
 }
```

More than one `AutoCloseable` resources can be used in `try-with-resources` statement separated by semicolon. Hence it is mandatory for `abc` and `pqr` objects to implement the `AutoCloseable` interface as shown below in the example. The resources created in the `try-with-resources` statement are closed in the reverse order of creation. We will elaborate these concepts in Example 7.7.

Example 7.7 AutoCloseable Resources and try-with-resource Statement

```
L1   class abc implements AutoCloseable
     {
L2     public void close()
       {
       System.out.println("Within close method of abc");
       }
     }
L3   class pqr implements AutoCloseable
     {
L4     public void close()
       {
            System.out.println("Within close method of pqr");
       }
     }
L5   class TestTryWithResources
     {
L6   public static void main(String args[])
     {
L7   try (abc a=new abc(); pqr p=new pqr())
       {
       System.out.println("Within try with resources block");
L8     throw new Exception();
       }
```

```
L9      catch(Exception e)
        {
        System.out.println("Within catch block");
        }
    }
    }
```

Output

```
D:\javabook\programs\chap 7\java TestTryWithResources
Within try with resources block
Within close method of pqr
Within close method of abc
Within catch block
```

Explanation

L1–4 All resources that need to be closed automatically after their use must implement the `AutoCloseable` interface and override the `close` method.
L5 Another class is created to test the `AutoCloseable` resources created above.
L6 `main` method declaration.
L7 `try-with-resources` statement is used to create two resources which will be automatically closed once the block exits by calling their respective `close` methods in reverse order of creation. The `close` method of `pqr` is called first and then the `close` method of `abc` (see output). Note that these two objects have already inherited the `AutoCloseable` interface otherwise a compile time error will be raised by the compiler.
L8 An explicit `Exception` is raised to show that the `close` methods are called irrespective of whether an exception occurs or not. In case an `Exception` is raised, the `close` methods are called prior to handling the `Exception` (see output).
L9 `catch` block is declared to handle the exception raised from the `try-with-resource` block.

Note It is not mandatory for a `try-with-resource` block to have a `catch` or `finally` block unlike the previous version of JDK. They are optional in Java 7 with a `try-with-resource` block.

7.2.6 Multi catch

Java 7 introduced the `multi catch` statement to catch multiple exception types using a single catch block. Example 7.3 showed the older ways of catching multiple exceptions using separate catch blocks. Assuming that `Exception1`, `Exception2`, and `Exception3` are belonging to different hierarchies and may be thrown from `try` block, they can be handled in a single `catch` block using the newer syntax for catching multiple exceptions as follows:

```
try
{
     // statements
}
catch (Exception1 | Exception2 | Exception3 e)
{
     // statements
}
```

So you might get the feeling that the `catch` block in Example 7.3 can be rearticulated as:

```
catch (ArithmeticException | Exception e)
{
     // statements
}
```

But the problem with the `catch` block above is that both `ArithmeticException` and `Exception` belong to the same hierarchy. (Actually every exception has branched out of `Exception`.) If the `catch` block is rearticulated as shown below, it compiles because now both exceptions belong to different inheritance hierarchy.

```
catch (ArithmeticException | NullPointerException | NumberFormatException e)
{
     // Statements
}
```

The benefit of using `multi catch` is that it results in more efficient byte code as you have just one `catch` block (instead of more as in the above case). Moreover same treatment can be applied to exceptions of different hierarchies. A way of applying different treatment while using `multi catch` syntax is by using `instanceof` operator as shown below. `instanceof` operator checks whether an instance is of a particular class and return true or false.

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException | NumberFormatException e)
{
     if(e instanceof ArithmeticException)
            System.out.println("Arithmetic Exception Handled: " +e);
     else if(e instanceof NumberFormatException)
            System.out.println("Exception Handled: " +e);
     else
            System.out.println(e);
}
```

Note In case the `multi catch` syntax is used, the parameter `e` is implicitly final.

7.2.7 Improved Exception Handling in Java 7

Prior to Java 7, a method can specify only those exceptions in the `throws` clause that have been specified in the `catch` clause while re-throwing exceptions from within `catch` block. But Java 7 onwards the `throws` can specify more refined exceptions to be rethrown. Suppose there are two user defined exceptions `Exception1` and `Exception2` which can be rethrown from within the `catch` block of a method. Prior to Java 7 only the exceptions specified in the `catch` block can be mentioned as argument to the `throws` keyword. Let us take an example to show this.

Example 7.8(a) Re-throwing an Exception

```
class Exception1 extends Exception { }
class Exception2 extends Exception { }
class DemoException{
L1   void throwException(int a, int b) throws Exception {
     try {
```

```
          if (a<b)
L2          throw new Exception1();
      else
L3          throw new Exception2();
L4    } catch (Exception e) {
L5        throw e;
      }
     }
    public static void main(String args[]) throws Exception
    {
      new DemoException().throwException(4,0);
    }
    }
```

The above method `throwException` could throw either `Exception1` (L2) or `Exception2` (L3) based on the value of `a` or `b`. Prior to Java 7, it was not possible to specify these exception types in the `throws` clause of the `throwException` method declaration (L1). The exception `e` is re-thrown from the `catch` block (L5) and as `e` is of type `Exception` so only `Exception` can be specified in the `throws` clause of method declaration on L1.

Java 7 onwards you can specify `Exception1` and `Exception2` in the `throws` clause of the `throwException` method declaration. The compiler deduces that the exceptions thrown by `throw e` (L5) must have come from the `try` block, and the exceptions thrown by the `try` block can be `Exception1` or `Exception2`. Although `e` is defined of type `Exception` (L4), the compiler can determine that `e` would be an instance of either `Exception1` or `Exception2`. Let us rephrase the method in the program.

Example 7.8(b) Re-throwing an Exception

```
    class DemoException{
L1    void throwException(int a, int b) throws Exception1, Exception2 {
      try {
          if (a<b)
L2          throw new Exception1();
      else
L3          throw new Exception2();
L4    } catch (Exception e) {
L5        throw e;
      }
    }
    public static void main(String args[]) throws Exception1,Exception2
    {
      new DemoException().throwException(4,0);
    }
    }
```

In other words, Java 7 onwards you can rethrow (L5) an exception that is a supertype (in our case it is `Exception`) of any of the types declared in the `throws` (i.e., `Exception1` and `Exception2`).

7.3 USER-DEFINED EXCEPTION

Java provides you with the opportunity to create your own exceptions, i.e., user-defined exceptions. The mandatory requirement is that the class should be a subclass of the `Exception` class. We will create a sample exception and use it in a different class and throw this particular exception on some particular condition.

Example 7.9 User-defined Exception

```
L1  class ExcepDemo extends Exception
    {
L2    ExcepDemo(String msg)
      {
L3          super(msg);
      }
L4    public String toString()
      {
L5          return "Exception in thread \"main\" ExcepDemo Exception:" +getMessage();
      }
    }
L6  class TestException
    {
L7    static void testException() throws ExcepDemo
      {
L8          throw new ExcepDemo("Testing User Defined Exception");
      }
L9  public static void main(String args[])
    {
L10         try
            {
L11         testException();
            }
L12         catch(ExcepDemo e)
            {
L13         System.out.println(e);
            }
    }}
```

Output

```
C:\javabook\programs\chap 7>java TestException
Exception in thread "main" ExcepDemo Exception: Testing User Defined Exception
```

Explanation

L1 To create your own exception, your class has to extend the `Exception` class as shown.

L2 Constructor for the exception subclass has been defined accepting a `String` argument.

L3 The `String` argument is passed to the superclass constructor using `super`. This argument can be retrieved using a method of the superclass, i.e., `getMessage()`.

L4 `toString` method has been overridden. This is automatically called when you print the object of the exception subclass.

L5 String is being returned concatenated with the output of the `getMessage()` function. It returns the string passed to the constructor of the superclass.

The user-defined class is ready and now we need a sample class to test it, so we created the `TestException` class.

L6 `TestException` class defined.
L7 `static` method declaring that it can throw `ExcepDemo` exception.
L8 Exception thrown using the keyword `throw`.
L9 `main` method declaration.
L10 `try` block defined.
L11 `testException()` method called.
L12 `catch` corresponding to `try` (L10).
L13 Prints the exception. `toString()` is called automatically, which returns the string `Exceptionin-thread "main" ExcepDemoException:` concatenated with the argument passed in the constructor of the `ExcepDemo` class in L8 (see output). This `String` is returned through the method `getMessage()`, defined in the `Throwable` class.

7.4 EXCEPTION ENCAPSULATION AND ENRICHMENT

Java 1.4 introduced *exception encapsulation* (chaining), which is the process of wrapping a caught exception in a different exception and throwing the wrapped exception. The `Throwable` class (parent class) has added a cause parameter in its constructors for wrapped exceptions and a `getCause()` method to return the wrapped exception. If you pass all your exception, your top level method might have to deal with a lot of exceptions; and declaring or handling exceptions in all the previous methods is a tedious task. The solution is to wrap exceptions and throw it. Wrapping is also used to abstract the details of implementation. You might not want your working details (including the exception that are thrown) to be known to others. Let us see how wrapping is done.

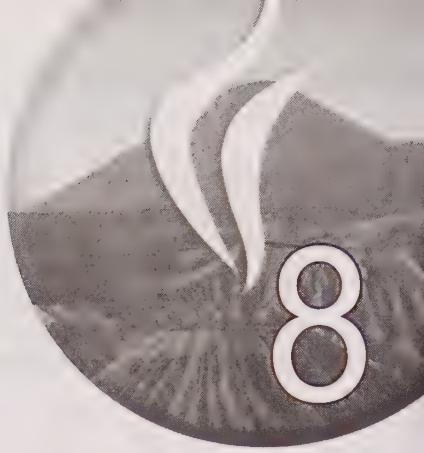
```
try{
     throw new InstantiationException();
}
catch(InstantiationException t)
  {
     // wrapping InstantiationException in ExcepDemo
     throw new ExcepDemo("Wrapped Instantiation Exception",t);
  }
```

Wrapping has some disadvantages also. It leads to long stack traces; one for each exception in the wrapping hierarchy. Secondly, due to wrapping, it becomes difficult to figure out the problem that led to exceptions.

The possible solution is *exception enrichment*. In exception enrichment, you do not wrap exceptions but add information to the already thrown exception and rethrow it, which leads to a single stack trace. Let us take an example to see exception enrichment.

Example 7.10 Exception Enrichment

```
L1   class ExcepDemo extends Exception{
       String message;
L2     ExcepDemo(String msg){
L3          message = msg;}
L4     public String toString(){
L5     return "Exception in thread \"main\" ExcepDemo Exception:" +message;
       }
L6     public void addInformation(String msg) {
L7          message += msg;
     }}
```



Multithreading in Java

A person who learns to juggle six balls will be more skilled than the person who never tries to juggle more than three.
Marilyn vos Savant

After reading this chapter, the readers will be able to

- know what are threads and how they can be implemented in Java
- understand how multiple threads can be created within a Java program
- understand different states of a thread in Java
- appreciate the `Thread` class of `java.lang` package
- understand how runnable interface is helpful in creating threads

8.1 INTRODUCTION

Until now, whatever programs we have discussed were sequential ones, i.e., each of them has a beginning, an execution sequence, and an end. While the program is being executed, at any point of time, there is a single line of execution. One thing that you must note that a thread in itself is not a program, as it cannot run on its own. However, it can be embedded with any other program.

Note A thread is a single sequential flow of control within a program.

The concept of single thread is quite simple to understand. Things become somewhat complex when there are multiple threads running simultaneously, each performing different tasks, within a single program. This can be enabled by multithreading, where you can write programs containing multiple paths of execution, running concurrently, within a single program. In other words, we can say that a single program having multiple threads, executing concurrently, can be termed as multithreaded program.

Let us go to the basics of multithreading, which is actually a form of multitasking. Multitasking can either be *process-based* or *thread-based*. If we assume programs as processes, then process-based multitasking is nothing but execution of more than one program concurrently. On the other hand, thread-based multitasking is executing a program having more than one thread, performing different tasks simultaneously. Processes are heavyweight tasks, while threads are lightweight tasks. In process-based multitasking, different processes are different programs, thus they share different address spaces. The context switching of CPU from one process to another

requires more overhead as different address spaces are involved in the same. On the contrary, in thread-based multitasking, different threads are part of the same program, thus they share the same address space and context switching of CPU occurs within the program, i.e., within the same address space. Obviously, this will require less overhead.

The objective of all forms of multitasking including multithreading is to utilize the idle time of the CPU. Ideally a CPU should always be processing something. The idle time of CPU can be minimized using multitasking.

Have you ever paid attention to one thing? When you prepare a document using a word processor program, the spelling can also be checked simultaneously. This is one such example of thread-based multitasking. While you type in the document, the CPU sits idle and waits for you to enter characters but because of thread-based multitasking, the word processor minimizes the CPU idle time somewhat by simultaneously involving the CPU in checking the spelling of the text. From now onwards, we will call thread-based multitasking as *multithreading*.

Note Multithreading enables programs to have more than one execution paths (separate) which execute concurrently. Each such path of execution is a thread. Through multithreading, efficient utilization of system resources can be achieved, such as maximum utilization of CPU cycles and minimizing idle time of CPU.

8.2 MULTITHREADING IN JAVA

Every program that we have been writing has at least one thread, i.e., the `main` thread. Whenever a program starts executing, the JVM is responsible for creating the main thread and calling the `main()` method, from within that thread. Alongside, many other invisible daemon threads responsible for supporting other activities of Java runtime such as finalization and garbage collection are also created.

Threads are executed by the processor according to the scheduling done by the Java Runtime System by assigning priority to every thread. It simply means, threads having higher priority are given preference for getting executed over the threads having lower priority.

When a Java program is executed, the JVM creates at least a single non-daemon thread (which calls the `main()` method of the corresponding class). A thread can either die naturally or be forced to die. The execution of the thread will go on until one of the following conditions occur:

- A thread dies naturally when it exits the `run()` method normally. The normal exit from `run()` means, the instructions of the `run()` has been processed completely.
- A thread can always be killed or interrupted by calling `interrupt()` method.

8.3 `java.lang.THREAD`

Creation of threads in Java is not as complex as the concept itself. There is a class named as `Thread` class, which belongs to the `java.lang` package, declared as,

```
public class Thread extends Object implements Runnable
```

This class encapsulates any thread of execution. Threads are created as the instance of this class, which contains `run()` methods in it. In fact the functionality of the thread can only be achieved by overriding this `run()` method. A typical `run()` would have the following structure:

```
public void run()
{
   ......
   // statement for implementing thread
   ......
}
```

Table 8.1 Methods of thread Class

Methods	Description
`static Thread currentThread()`	Returns a reference to the currently executing thread.
`static intactiveCount()`	Returns the current number of active threads.
`long getID()`	Returns the identification of thread.
`final String getName()`	Returns the thread's name.
`final void join()`	Waits for a thread to terminate.
`void join (long m)`	Waits at the most for 'm' milliseconds for the thread to die.
`void join (long m, int n)`	Waits at the most for 'm' milliseconds and 'n' nanoseconds for the thread to die.
`void run()`	Entry point for the thread.
`final void setDaemon(boolean how)`	If how is true, the invoking thread is set to daemon status.
`boolean isInterrupted()`	Returns true if the thread on which it is called has been interrupted.
`final boolean isDaemon()`	Returns true if the invoking thread is a daemon thread.
`final boolean isAlive()`	Returns boolean value stating whether a thread is still running.
`void interrupt()`	Interrupts a thread.
`static boolean holdsLock(Object anyObj)`	Returns true if the invoking thread holds the lock on `anyObj`.
`Thread.State getState()`	Returns the current state of the thread.
`final int getPriority()`	Returns the priority of the thread.
`static boolean interrupted()`	Returns true if the invoking thread has been interrupted.
`final void setName(String thrdName)`	Sets a thread's name to `thrdName`.
`final void setPriority(int newPriority)`	Sets a thread's priority to `newPriority`.
`static void sleep(long milliseconds)`	Suspends a thread for a specified period of milliseconds.
`void start()`	Starts a thread by calling its `run()` method.
`void destroy()`	Destroys the thread, without any clean up.
`static int enumerate (Thread[] thrdArray)`	Copies into the specified array, every active thread of thread's group and sub group.
`static void yield()`	Cause the current executing thread to pause and allow the other threads to execute.

This method is automatically invoked when a thread object is created and initiated using the `start()` method. Some of the methods belonging to the `Thread` class, which help in manipulating thread instances, are shown in Table 8.1.

Apart from these, some constructors are also defined in the `Thread` class. These constructors can be classified in two different categories. We will discuss one category here and the other in the next section when we will discuss about `Runnable` interface. The constructors responsible for creating threads are

1. `Thread()`
2. `Thread(String threadName)`
3. `Thread(ThreadGroup threadGroup, String threadName)`

In the first constructor, you can see that there are no arguments, which simply means it uses the default name and the thread group. In the second constructor, the name of the constructor can be specified as `String`. While in the third, you can specify the thread group and thread name.

8.4 MAIN THREAD

Even if a thread is not created by a programmer, every Java program has a thread, the `main` thread. When a normal Java program starts executing, the JVM creates the `main` thread and calls the program's `main()`method from within that thread. Apart from this, the JVM also creates some invisible threads, which are important for its housekeeping tasks such as, threads taking care of garbage collection and threads responsible for object finalization. The `main` thread spawns the other threads. These spawned threads are called *child threads*. This `main` thread is always the last to finish executing because it is responsible for releasing the resources used during the program execution, such as network connections.

As a programmer, you can always take control of the `main` (or any other) thread. For this, a static method, `currentThread()`, is used to return a reference to the current thread. The `main` thread can be controlled by this reference only.

Now let us put these into practicality by creating a reference to the `main` thread. We could also change the name of the `main` thread from `main` to any new name. The following piece of code serves the purpose for you.

Example 8.1 Renaming a Thread

```
L1  class MainThreadDemo {
L2    public static void main (String args[] ) {
L3    Thread threadObj = Thread.currentThread();
L4    System.out.println("Current thread: " +threadObj);
L5    threadObj.setName("New Thread");
L6    System.out.println("Renamed Thread: " +threadObj);
L7  } }
```

Output

```
Current thread: Thread[main, 5, main]
Renamed Thread: Thread[New Thread, 5, main]
```

Explanation

L1 Class `MainThreadDemo` declared.

L2 `main` method declared.

L3 A reference to the current `Thread` is returned and is stored in the `threadObj`. Here the current thread is the `main` thread itself. The reference is declared by specifying the name of the class, i.e., `Thread` class in this case followed by the name for the reference, which is done as in the following line of code:

```
Thread threadObj
```

We acquire a reference to the main thread by calling the static method `currentThread()` of the `Thread` class using the following method call:

```
Thread.currentThread()
```

The reference to the current thread object (i.e., `main`) is returned by the `currentThread()`method and stored in the reference previously declared.

L4 The thread object (i.e., `main`) is passed to the `println` method. The `toString()` method of the `Thread` class is called by default, which displays the first line of the output.

L5 The `setName()` method of `Thread` class is used to change the name of the `Thread`. This example uses the `setName()` method to change the main thread's name from `main` to `New Thread`.

If we see the output now, the information within the square brackets is the signature of the thread. The first element in the bracket is the name of the thread. The second element signifies the thread priority (explained later in the Chapter under the topic *thread priority*). The range for setting the priority can be between 1 and 10; 1 for lowest priority 10 for highest priority and 5 for normal priority. The last element in the bracket is the group name for threads to which the thread belongs. The state of collection of threads can be controlled by a data structure, known as *thread group*. The thread group is automatically handled by the Java runtime system.

8.5 CREATION OF NEW THREADS

Once we have mentioned some of the methods and constructors of the `Thread` class, we can concentrate on different ways to create a new thread:

- By inheriting the `Thread` class
- By implementing the `Runnable` interface

8.5.1 By Inheriting the Thread Class

Threads can be created by inheriting the `java.lang.Thread` class. All the thread methods belonging to the `Thread` class can be used in the program because of the extension (inheritance).

Steps to be followed for thread creation:

- Declare your own class as extending the `Thread` class.
- Override the `run()` method, which constitutes the body of the thread.
- Create the thread object and use the `start()` method to initiate the thread execution.

Let us elaborate these steps in detail.

Declaring a Class Any new class can be declared to extend the `Thread` class, thus inheriting all the functionalities of the `Thread` class.

```
classNewThread extends Thread
{
   ......
   ......
   ......
}
```

Here, we have a new type of thread, named as 'NewThread'.

Overriding the run() Method The run() method has to be overridden by writing codes required for the thread. The thread behaves as per this code segment. A typical run() method would look like

```
public void run( )
{
   ......
   //code segment providing the functionality of thread
   .....
}
```

Starting New Thread The third part talks about the start() method, which is required to create and initiate an instance of our Thread class. The following piece of code is responsible for the same:

```
newThread thread1 = new newThread();
thread1.start();
```

The first line creates an instance of the class NewThread, where the object is just created. The thread is in *newborn state*. Second line, which calls the start() method, moves the thread to *runnable state*, where the Java Runtime will schedule the thread to run by invoking the run() method. Now the thread is said to be in *running state*.

Example 8.2 Creating a Thread Using the Thread Class

```
L1  class ThreadOne extends Thread {
L2  public void run(){
L3  try {
L4    for(int i = 1; i<= 5; i++) {
L5    System.out.println("\tFrom child thread 1 : i =" +i);
L6    Thread.sleep(600);
L7  }
L8  } catch(InterruptedException e){
L9  System.out.println("child thread1 interrupted");
L10 }
L11 System.out.println("Exit from child thread 1");
L12 }
L13 }
L14 class ThreadTwo extends Thread{
L15 public void run(){
L16 try {
L17   for(int j = 1; j <= 5; j++){
L18   System.out.println("\t From child thread 2 : j =" +j);
```

```
L19     Thread.sleep(400);
L20   }
L21   } catch(InterruptedException e){
L22   System.out.println("child thread 2 interrupted");
L23   }
L24   System.out.println("Exit from child thread 2");
L25   }
L26   }
L27   class ThreadThree extends Thread {
L27   public void run(){
L28   try {
L29     for(int k = 1; k <= 5; k++) {
L30     System.out.println("\tFrom child thread 3 : k =" +k);
L31     Thread.sleep(800);
L32   }
L33   } catch(InterruptedException e){
L34   System.out.println("child thread 3 interrupted");
L35   }
L36   System.out.println("Exit from child thread 3");
L37   }
L38   }
L39   class ThreadDemo {
L40     public static void main(String arg[]) {
L41     ThreadOne a = new ThreadOne();
L42     a.start();
L43     ThreadTwo b = new ThreadTwo();
L44     b.start();
L45     ThreadThree c = new ThreadThree();
L46     c.start();
L47     try {
L48     for(int m=1; m<=5; m++){
L49     System.out.println("\t From Main Thread : m =" +m);
L50     Thread.sleep(1200);
L51     }
L52   } catch (InterruptedException e) {
L53   System.out.println("Main interrupted");}
L54   System.out.println("Exit form main thread");
L55   }
L56   }
```

Output

```
From child thread 1 :i =1
From Main Thread : m =1
From child thread 2 : j =1
From child thread 3 : k =1
From child thread 2 : j =2
From child thread 1 :i =2
From child thread 3 : k =2
From child thread 2 : j =3
From Main Thread : m =2
From child thread 1 :i =3
```

```
   From child thread 2 : j =4
   From child thread 3 : k =3
   From child thread 2 : j =5
   From child thread 1 :i =4
Exit from child thread 2
   From Main Thread : m =3
   From child thread 1 :i =5
   From child thread 3 : k =4
Exit from child thread 1
   From child thread 3 : k =5
   From Main Thread : m =4
Exit from child thread 3
   From Main Thread : m =5
Exit from main thread
```

Explanation

L1 Class `ThreadOne` extends the `Thread` class, thus inheriting all the functions and members of the `Thread` class.

L2–7 `run()` method, returning void is overridden. The `for` loop incrementing the counter variable, `i`, is looped 5 times (L3). Each value of `i` is displayed on the screen (L4) and before moving to the next value of `i`, the thread sleeps for 0.6 seconds (L6). `Thread.sleep()` method throws an exception, `InterruptedException`, so it should be within a `try…catch` block.

L14 Just like the class `ThreadOne`, a new class `ThreadTwo`, extending the `Thread` class is declared.

L15–20 `run()` method responsible for providing the functionality of the thread of this class is overridden. The code of this method is similar to that of the `run()`, explained in the previous paragraph.

L27 Third class, `ThreadThree`, extending the `Thread` class is declared.

L28–36 `run()` method for the third class' thread is implemented, similar to the previously explained `run()` methods.

L39 Class `ThreadDemo` encapsulating the `main()` method is declared. This class, which acts as the main thread, is responsible for spawning the other three child threads.

L40 `main()` method declared.

L41 Reference for `ThreadOne` class is created and stored in `a`.

L42 The `start()` method is invoked on the thread object `a`. This method puts the thread in a ready-to execute state. As soon as the CPU is allocated to the thread by the thread scheduler, the `run()` method for the thread is called automatically. As you can see in the example also, the `run()`method is not called explicitly.

L43–46 Just like creating the object for `ThreadOne`, we create the reference objects for `ThreadTwo` and `ThreadThree` and store them in `b` and `c`, respectively. L44 is responsible for starting the second thread pertaining to `ThreadTwo` class and L46 is responsible for starting the third thread pertaining to `ThreadThree` class, thus resulting in invocation of the corresponding `run()` methods.

L47–55 Certain functionalities, similar to the functionalities of the above child threads, are provided inside the main thread also. It has been made to sleep for 1.2 seconds (L50), which has been kept more than the three child threads, so that the `main` thread completes its execution at last, otherwise there is always a possibility for the system to get hung.

8.5.2 Implementing the `Runnable` Interface

We have already mentioned that there can be two ways for implementing threads. First method has already been discussed in the previous section. Now let us talk about the second way, i.e.,

by implementing the `Runnable` interface. Before taking on the second method of implementing `Runnable` interface, we must know the ins and outs of this interface. It is actually implemented by class `Thread` in the package `java.lang`. This interface is declared public as,

```
public interface Runnable
```

The interface needs to be implemented by any class whose instance is to be executed by a thread. The implementing class must also override a method named as `run()`, defined as the only method in the `Runnable` interface as,

```
public void run( )
{
   ......
   ......
}
```

The object's `run()` method is called automatically whenever the thread is scheduled for execution by the thread scheduler. The functionality of the thread depends on the code written within this `run()` method. One thing worth noting is that other methods can be called from within `run()`. Not only this, use of other classes and declaration of variables, just like the `main` thread, are also possible inside `run()`. The thread will stop as soon as the `run()` exits.

The question that arises here is, when and how shall we resort to the second method? The approach to be undertaken is dependent on the requirement of the class. If the class requires inheriting any other class, then obviously the `Thread` class cannot be inherited, as multiple inheritance is not allowed in Java. So the obvious solution in this case is to use the interface, i.e., `Runnable`. [Remember: any numbers of interfaces can be inherited by a class.]

Some other constructors belonging to the `Thread` class are worth mentioning here, as these can be used while creating thread using `Runnable` interface.

- `Thread(Runnable threadObj)`
- `Thread(Runnable threadObj, String threadName)`
- `Thread(ThreadGroup threadGroup, Runnable threadObj)`
- `Thread(ThreadGroup threadGroup, Runnable threadObj, String threadName)`

The above `threadObj` is a reference to an instance of a class that implements the `Runnable` interface and overrides the `run()` method. This defines where the execution will begin. As mentioned earlier, the object's `run()` method's code is responsible for giving the functionality to the new thread. `threadName` is the name of the thread. In case no name is passed externally to the constructors, the JVM is automatically going to name it. The third argument, `threadGroup` is the group name to which the thread belongs. If no thread group is externally specified, the group is determined by the security managing component or the group is set to the same group, which the invoking thread is a part of.

Once a class that implements the `Runnable` interface is created, an object of the `Thread` class must be instantiated from within that class. In Example 8.3, we are going to use the second constructor described in the previous paragraph, i.e.,

```
Thread (Runnable threadObj, String threadName)
```

Even if a thread is created, it will not start executing unless the `start()` method of the `Thread` class is called.

Example 8.3 Creating a Thread Using Runnable Interface

```
L1   class ThreadChild implements Runnable {
L2   ThreadChild() {
L3   Thread t = new Thread (this, "Example Thread");
L4   System.out.println("Detail of child thread :" +t);
L5   t.start();
L6   }
L7   public void run(){
L8   try {
L9     for(int i = 1; i<= 5; i++) {
L10    System.out.println("\tFrom child thread 1 : i =" +i);
L11    Thread.sleep(500);
L12  }
L13  } catch(InterruptedException e) {
L14  System.out.println("child Thread 1 interrupted");
L15  }
L16  System.out.println("Exit from child Thread 1");
L17  }
L18  }
L19  class ThreadDemo2 {
L20    public static void main(String args[]) {
L21    new ThreadChild();
L22    try{
L23    for(int m=1; m<=5; m++) {
L24    System.out.println("\tFrom Main Thread : m =" +m);
L25    Thread.sleep(1000);
L26  }
L27  } catch(InterruptedException e){
L28  System.out.println("Main interrupted");
L29  }
L30  System.out.println("Exit from main thread");
L31  }
L32  }
```

Output

```
   Detail of child thread :Thread[Example Thread,5,main]
   From Main Thread : m = 1
   From child thread 1 :i = 1
   From child thread 1 :i = 2
   From child thread 1 :i = 3
   From Main Thread : m = 2
   From child thread 1 :i = 4
   From Main Thread : m = 3
   From child thread 1 :i = 5
Exit from child Thread 1
   From Main Thread : m = 4
   From Main Thread : m = 5
Exit from main thread
```

Explanation

L1 A class, `ThreadChild`, implementing the `Runnable` interface is declared. This class is responsible for the creation of the child thread, spawning out of the main thread.

L2 A constructor, `ThreadChild()`, is declared.

L3 Inside `ThreadChild()` constructor, a thread object `t` is created. Passing `this` as the first argument shows that the current class has inherited the `Runnable` interface. It is this class' object that will tell what the thread is going to perform because it has overridden the `run()` method.

L5 The `start()` method is called, which starts the execution of the thread by invoking the `run()`.

L7–15 `run()` is declared (L7) . It will be called automatically whenever this thread is scheduled by the thread scheduler. [Remember: explicit call to `run()`will invoke it from the caller thread rather than its own thread]. A `for` loop is declared to loop five times to display the numbers from 1 to 5. After displaying each number on the screen, the thread sleeps for half a second. You can see the `try...catch` block, which is placed to catch the `InterruptedException` thrown by the `sleep()` method.

L16 We intend to display the exit of the child thread created inside the `ThreadChild` class.

L19 Class `ThreadDemo2` containing the `main` thread is declared (L31).

L20–26 The `main` method is declared (L20). The constructor `ThreadChild()` is instantiated (L21). This instantiation invokes the constructor declared between L2–6. The `try...catch` block between L22–26 constitutes the body of the `main` thread. A `for` loop is declared to loop five times and it displays the number from 1 to 5. After displaying each number on the screen, it sleeps for one second.

8.6 `Thread.State` IN JAVA

In the process-based multitasking, the operating system manages the context switching between programs based on the time slice provided for each program segment, but in case of multithreading, Java Runtime manages threads. Here, execution of one thread stops while the other continues, as the switching takes place within the same program amongst different threads. Before Java 5.0, Java had thread states similar to the thread states of an operating system. Java 5.0 came up with a static nested class named as `State`, which is made a part of the `Thread` class. This `Thread` class is made to inherit the abstract class `Enum`. This class `Enum` is a common base class of all Java language enumeration types. In other words, `Thread.State` is actually an enumeration type declared as follows:

```
public static enum Thread.State extends Enum
```

The enumeration `Thread.State` has the possible states of a Java thread in the underlying JVM. At any time, a thread is said to be in one of the states mentioned below. Figure 8.1 shows the various states in which a Java thread exists during its life. Some of the methods responsible for the transition from one state to another are also shown. Obviously, the diagram is not exhaustive, as it does not mention all such responsible methods. It is just an overview of a thread's life.

New In this state, a new thread is created but not started. The following line of code is responsible for the same (assuming `ThreadDemo` class has inherited the `Thread` class or `Runnable` interface):

```
Thread threadObj = new ThreadDemo();
```

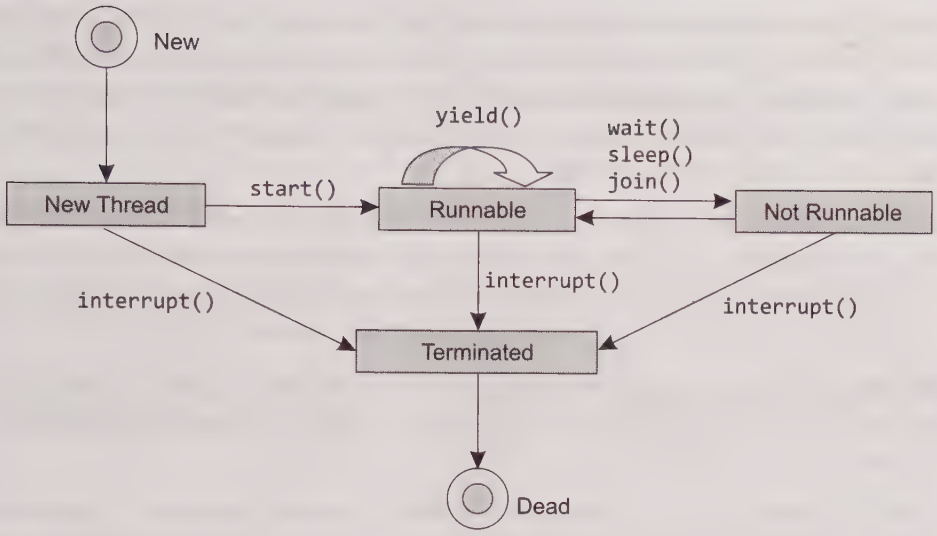


Fig. 8.1 Thread States

The above statement is responsible for creating a new `Thread` object. In 'New' state, no system resource (such as CPU) is allotted to the newly born `Thread` object. From this state, the thread can either be started (by using `start()` of `Thread` class) or stopped (by using `interrupt()` of `Thread` class), thus moving to 'Runnable' or 'Terminated' state, respectively. No other method apart from `start()` and `interrupt()` can be called from this state and if tried to do so, it would cause an exception, `IllegalThreadStateException`.

Runnable In this state, a thread is ready for execution by the JVM. It represents the running state of the thread, as well. Ready state of a thread can be defined as it is ready for execution but it might be in the queue, waiting for the operating system to provide it the required resource, like processor. Once a thread is actually being executed by the processor then it is termed as "Running". From 'New' state the thread might move to the 'Runnable' state on execution of the following statements:

```
Thread threadObj = new ThreadDemo();
threadObj.start();
```

As soon as `start()` is called, the thread is allotted the system resource as per the scheduling done by the Java Runtime Environment. Now the thread has entered into the runnable state. In Fig. 8.1, no differentiation is made between a running thread and a runnable thread. Even the running threads are made a part of the runnable state. But there is a difference between the two. A running thread is the one which is being executed by the processor. Such a thread can be called as the *current thread*. Runnable threads are those which are not actually running, but are scheduled in queue to get the processor. The scheduling scheme, under which all the

runnable threads are prioritized for sharing the processor, is implemented by the Java Runtime system. However, when a thread moves to 'Running' from 'Runnable', the instructions of the `run()` method are being executed sequentially. During this phase the processor can be forced to relinquish its control over the thread, thus forcing it to be a part of the queue again by the use of `yield()` method as shown in Fig. 8.1.

Not Runnable From runnable state, a thread might move to the not runnable state, as shown in Fig. 8.1. This state is just a hypothetical state used by us to categorize the three valid states of Java. A thread which is in any of these three states can be assumed to be in 'not runnable' state. These three states are WAITING, TIMED_WAITING, and BLOCKED.

Waiting In this state, a thread is waiting indefinitely for another thread to perform a particular action (i.e., notify). Threads can move into this state either by calling the methods `Object.wait()` (without time out) or `Thread.join()` (without time out).

Timed_Waiting In this state, the thread is waiting for another thread to perform an action (notify) up to a specified waiting time. A thread can get into this state by calling either of these methods: `Thread.sleep()`, `Object.wait()`, and `Thread.join()` (all these methods should be called with time out specified).

Blocked In this state, a resource cannot be accessed because it is being used by another thread. A thread can get into this state by calling `Object.wait()` method.

Before proceeding further, we must discuss the concept of monitors in Java. This is taken up in greater detail in Section 8.8. Monitor is an object that is a mutually exclusive lock on the resource to be accessed. A monitor can be owned by only one thread at a time. When a thread calls `Object.wait()` method, it releases all the acquired monitors and is put into WAITING state, until some other thread enters the same monitor and calls `notify()/notifyAll()`. When `notify()` is called, it wakes up a thread that called `wait()` on the same object. The method `notifyAll()` will wake up all the threads that called `wait()` on the same object. The difference between two methods is that, if `notify()` is used, then only one thread (selected by the JVM scheduler) is granted the monitor and all other threads are put into BLOCKED state, whereas if you use `notifyAll()`, it wakes up all the threads and puts them into ready state. The threads that can execute, start executing, and the rest move into the waiting state.The three methods mentioned above are `final` methods of the `ObjectClass`, so all classes have them.

```
final void wait() throws InterruptedException;
final void notify()
final void notifyAll()
```

Additional form of `wait()` where time can be specified for the thread to wait for that period, is also available. It puts the thread in TIMED_WAITING state. We can easily figure out that a WAITING state thread will always be dependent on an action performed by some other thread, whereas a thread in TIMED_WAITING is not completely dependent on an action performed by

some other thread, as in this case, the wait ends automatically after the completion of the time out period. Similarly, if a thread has put itself into WAITING state by calling `Thread.join()` method, then it will keep waiting until the specified thread terminates or the specified time elapses. There seems to be no difference between `sleep()` and `wait()` as both of them do the same job of making a thread wait for a specified time. The differences between the start methods have been specified in Table 8.2.

Table 8.2 Difference between `wait()` and `sleep()`

`Object.wait()`	`Thread.sleep()`
`wait()` belongs to `Object` class	`sleep()` belongs to `Thread` class.
It can only be used from within the synchronized method or statements.	It can be used from outside the synchronized methods and statements.
Wait state can be terminated by calling `Object.notify()` method.	The sleep state can be terminated by invoking `interrupt()` method of the thread instance.
`Object.wait()` is used in concurrent thread access codes only.	`Thread.sleep(int ms)` is used wherever and whenever required method.
It stops the current thread execution and releases the lock of the object. Now other threads can use this released object.	`Thread.sleep(int ms)` causes the current thread to suspend execution for a specified number of milliseconds. This can let other threads to use the resources being held by the previous thread

Terminated

This state is reached when the thread has finished its execution. A thread can move to 'Terminated' state, from any of the above mentioned states. In this state, the thread is dead. The death of a thread can either be natural or forceful. A thread dies naturally when it exits `run()` normally. The normal exit from `run()` means the instructions of the `run()` has been processed completely. For example, the `for` loop in the following method is a finite loop which would iterate 5 times (i.e., from 1 to 5) and then exit.

```
public void run(){
for(int i = 1; i<= 5; i++)
{
   System.out.println("i =" +i);
   Thread.sleep(500);
}
}
```

A thread with the above `run()` method will die naturally after the last statement of `run()` completes. A thread can always be interrupted by using `interrupt()`. The following block of code does the job of killing a thread by calling `interrupt()` method.

We can know the state of a particular thread by using `getState()` method. The following program shows the usage of `getState()` and `interrupt()` methods.

Example 8.4 Using interrupt() and getState()

```
L1   class ThreadInterrupt extends Thread {
L2   boolean interrupt = false;
L3   String name;
L4   ThreadInterrupt(String n){
L5   super(n);
L6   name = n;
     }
L7   public void run(){
L8   while (!interrupt){
L9   System.out.println("Thread running: " +name+ " state: " +getState());
L10  try{
L11  Thread.sleep(1000);
L12  } catch(InterruptedException e){
L13  System.out.println("Thread Interrupted:" +name + "state:" +getState());
     }
     }
L14  System.out.println("Thread exiting under request: "+name + "state: "+getState());
     }
L15  public static void main(String args[]) throws Exception {
L16  ThreadInterrupt thread = new ThreadInterrupt("InterruptExample");
L17  System.out.println("Starting Thread: " +thread.name + "state: " +thread.getState());
L18  thread.start();
L19  Thread.sleep(3000);
L20  System.out.println("Stopping Thread: " +thread.name + "state: " +thread.getState());
L21  thread.interrupt = true;
L22  thread.interrupt();
L23  System.out.println(thread.name +" state: "+thread.getState());
L24  Thread.sleep(3000);
L25  System.out.println("Exiting application state: " +thread.getState());
L26  System.exit(0);
     }}
```

Output

```
C:\javabook\programs\chap 8>java ThreadInterrupt
Starting Thread: InterruptExample state: NEW
Thread running: InterruptExample state: RUNNABLE
Thread running: InterruptExample state: RUNNABLE
Thread running: InterruptExample state: RUNNABLE
Stopping Thread: InterruptExample state: TIMED_WAITING
InterruptExample state: TIMED_WAITING
Thread Interrupted: InterruptExample state: RUNNABLE
Thread exiting under request: InterruptExamplestate : RUNNABLE
Exiting application state: TERMINATED
```

Explanation

L1 A thread class is created.

L2 A Boolean variable `interrupt` (similar to a flag) has been defined to check the thread interrupted status.

L3 A string instance variable `name` has been defined to assign a name to the thread.

L4–6 Constructor for the class has been defined. This constructor sets the name of the thread by using the `super` constructor call and assigns the argument value to the string variable `name`.

L7 The `run()` method which states what the thread has to perform is overridden.

L8 It checks the status of the `interrupt` flag and if it is false, the `while` loop keeps on executing.

L9 It is a print statement that displays the name of the thread along with its state. The current state of the thread is obtained using the method `getState()`. The state of a thread if enquired from a run method will always be RUNNABLE as the run method is only executed when a thread is executing. (See output)

L10–13 A `try...catch` block has been defined because the thread is made to sleep for a second using `Thread.sleep(1000)` method and the `sleep` method may throw `InterruptedException`, if interrupted. So it has to be caught. The `throws` keyword cannot be used with the `run` method as the method is overridden and the parent interface (`Runnable`) does not mention any `throws` clause with the definition of the `run` method. So during overriding, the definition of the method cannot be changed. If an exception is generated, it is caught by the catch defined in L12 and L13 is executed, which displays `Thread Interrupted:` followed by the name of the thread and its state (i.e. RUNNABLE).

L14 It is the final print statement in the `run` method that displays the thread exiting under request followed by the name of the thread and its state, i.e., RUNNABLE. This line will be executed after the while loop exits.

L15 It defines the starting point for the execution of the program, i.e., the `main` method.

L16 A new thread is created and the name for the thread is passed as an argument in the constructor of the thread.

L17 Shows a print statement that displays `Starting Thread` followed by the name of the thread and its state. The thread has just been created, so its state will be NEW.

L18 Starts the thread by using the `start()` method on the newly created thread instance in L16. The `run` method for this thread will be called automatically as the thread is scheduled for execution (L7). As the interrupt flag is initially false, the while loop in the run method will execute and L9 will keep on executing.

L19 The `main` thread is made to sleep for 3 seconds. So the other thread `InterruptedExample` will keep on executing and sleeping (1 second only).

L20 Shows a print statement that displays the `Stopping Thread:` followed by the name of the thread and its state. If you see the output, the state of the thread displayed is TIMED_WAITING because the `InterruptedExample` thread is sleeping for 1 second and the main thread is executing at this moment.

L21–22 The `interrupt` boolean variable is set to `true` and the `InterruptedExample` thread is interrupted using `interrupt()`. Note that the `InterruptedThread` is sleeping and if a thread is interrupted while it is sleeping, an `InterruptedException` is generated. This exception will be caught at the `catch` defined in L12 and this block will execute as soon as the thread regains CPU, in other words is scheduled by the scheduler.

L23 Shows a print statement that displays the name of the thread and its state. If you see the output, the state of the thread is still TIMED_WAITING because the `InterruptedExample` thread is still not allowed to execute as the `main` thread is executing.

L24 The `main` thread is deliberately made to sleep to allow the other thread to execute. At this stage, the control passes to `catch` in L12. L13 executes followed by L14 and then the `run` method exits.

L25 Shows a print statement that displays Exiting Application followed by the state of the child thread (created from main) i.e., `InterruptedExample`. The `run` method for the thread `InterruptedExample` has exited, so the state is now TERMINTAED (see output).

L26 The application is terminated using the `exit` method of the `System` class.

There was one more method, `stop()`, which has now been deprecated, used to terminate a thread. The reason for this deprecation is that it throws a `ThreadDeath` object at the thread to kill it. Apart from this, calling `stop()` method results in sudden termination of thread's `run()` method, which might lead to the results achieved by the thread program in inconsistent or undesirable state.

8.7 THREAD PRIORITY

Each thread has a set priority, which helps the scheduler to decide the order of sequence of thread execution, i.e., when should which thread run? By default the threads created, carry the same priority, due to which the Java scheduler schedules them for the processor on first-come-first-serve basis. It is to be noted that Java follows preemptive scheduling policy, just like an operating system. When a high priority thread becomes ready for execution, the currently executing low priority will be stopped. On the contrary, a low priority thread cannot preempt a currently running high priority thread. It has to wait until the high priority thread is dead or blocked because of some reason or the other. The reasons for this can be any of the following:

- Thread stops as soon as it exits `run()`
- It sleeps (by using `sleep()`)
- It waits (by using `wait()` or `join()`)

Once it resumes from the blocked state, it will again preempt the low priority thread to which it had relinquished its control earlier, thus forcing the low priority to move to the runnable state from the running state.

Note Higher priority threads will always preempt the lower priority threads. Actually it depends on how the priorities of threads set by the JVM are mapped to the operating system. It might happen that a higher priority might not be considered higher by the operating system. So this actually depends on the operating system and it varies from OS to another.

As shown in Table 8.3, the `Thread` class has a method `setPriority()`, responsible for setting the priority of the thread by programmer. The signature of the method is

```
final void setPriority(int x)
```

where `x` specifies the value used to signify the thread's priority. `Thread` class defines several predefined priority constants (as static final variables) as shown in Table 8.3.

Table 8.3 Priority Constants and their Corresponding Value for Threads

Constant	Value	Meaning
MIN_PRIORITY	1	Max priority a thread can have
NORM_PRIORITY	5	Default priority a thread can have
MAX_PRIORITY	10	Min priority a thread can have

From the above table, it is clear that priority can be set in the form of values between 1 and 10. If this priority is not externally assigned, by default it is set to NORM_PRIORITY, i.e., 5.

A thread's current priority can be obtained by the `getPriority()` of the `thread` class, which returns an integer value.

finalintgetPriority()

Here is an example that shows the use of priority constants and `getPriority()` method.

Example 8.5 Setting and Getting Priorities of Threads

```
L1  class ThreadOne extends Thread {
L2  public void run(){
L3  try {
L4    for(int i = 1; i<= 5; i++){
L5         System.out.println("\tFrom child thread 1 : i =" +i);
L6         Thread.sleep(500);
L7       }
L8  } catch(InterruptedException e){
L9    System.out.println("child Thraed 1 interrupted");
L10 }
L11 System.out.println("Exit from child Thread 1");
L12 }
L13 }
L14 class ThreadTwo extends Thread {
L15 public void run() {
L16 try {
L17   for(int j = 1; j <= 5; j++) {
L18        System.out.println("\tFrom child thread 2 : j =" +j);
L19        Thread.sleep(500);
L20     }
L21 } catch (InterruptedException e) {
L22   System.out.println("child thread 2 interrupted");
L23 }
L24 System.out.println("Exit from child thraed 2");
L25 }
L26 }
L27 class ThreadThree extends Thread {
L28 public void run() {
L29 try{
L30   for(int k = 1; k <= 5; k++) {
L31        System.out.println("\tFrom child thread 3 : k =" +k);
L32        Thread.sleep(500);
L33     }
L34 } catch (InterruptedException e) {
L35   System.out.println("child thread 3 interrupted");
L36 }
L37 System.out.println("Exit from child thread 3");
L38 }
L39 }
L40 class ThreadPriority {
L41 public static void main(String args[]) {
L42   ThreadOne a = new ThreadOne();
L43   ThreadTwo b = new ThreadTwo();
L44   ThreadThree c = new ThreadThree();
L45   System.out.println("Default Priority for thread 1:" +a.getPriority());
L46   System.out.println("Default Priority for thread 2:" +b.getPriority());
```

```
L47     System.out.println("Default Priority for thread 3:" +c.getPriority());
L48     System.out.println(" ");
L49     a.setPriority(Thread.MIN_PRIORITY);
L50     b.setPriority(Thread.NORM_PRIORITY);
L51     c.setPriority(Thread.MAX_PRIORITY);
L52     System.out.println("Priority set for thread 1 :" +a.getPriority());
L53     System.out.println("Priority set for thread 2 :" +b.getPriority());
L54     System.out.println("Priority set for thread 3 :" +c.getPriority());
L55     System.out.println(" ");
L56     System.out.println("All the Three threads start from here");
L57     a.start();
L58     b.start();
L59     c.start();
L60   }
L61   }
```

Output

```
Default Priority for thread 1 :5
Default Priority for thread 2 :5
Default Priority for thread 3 :5
New Priority set for thread 1 :1
New Priority set for thread 2 :5
New Priority set for thread 3 :10
All the Three threads start
  from here From child thread
  3 : k = 1 From child thread
  2 : j = 1 From child thread
  1 : i = 1 From child thread
  3 : k = 2 From child thread
  2 : j = 2 From child thread
  1 : i = 2 From child thread
  3 : k = 3 From child thread
  2 : j = 3 From child thread
  1 : i = 3 From child thread
  3 : k = 4 From child thread
  2 : j = 4 From child thread
  1 : i = 4 From child thread
  3 : k = 5 From child thread
  2 : j = 5 From child thread
  1 : i = 5
Exit from child thread 3
Exit from child thread 2
Exit from child thread 1
```

Explanation

L1–39 These lines have the details about declaring three different classes, `ThreadOne`, `ThreadTwo`, and `ThreadThree`, each extending the `Thread` class. Each of these child thread classes have already been explained in Example 8.2.

L40 Main thread class `threadPriority` is declared.

L41 The `main()`method is declared.

L42–45 Reference objects for the three child thread classes are created and stored in `a`, `b`, and `c`.

L45 Default priority for child thread one is displayed. You can see the use of `getPriority()` method, which has been called using the object of `ThreadOne` class. This method returns the current priority of the thread pertaining to `ThreadOne` class.
L46–47 Just like child thread one, the current priority of child thread two and child thread three is displayed.
L49 Object `a` of child thread class, `ThreadOne` invokes its `setPriority()` method to set the priority for the execution of threads, so as to schedule the threads externally. You can see that the priority for thread one is set to minimum priority, i.e., 1, by passing it as argument to `setPriority()`.
L50 Similar to thread one, the object of child thread two invokes its `setPriority()` method to set the priority of the thread to normal priority, i.e., 5.
L51 The object of child thread three is used to call its `setPriority()` method to set the priority of the thread to highest priority, i.e., 10.
L52–54 New priorities of the three children threads are displayed. Once more, `getPriority()` method can prove to be instrumental for the purpose of obtaining the newly set priorities of the threads.
L57–59 From within the main thread, all the three threads started using the `start()`methods corresponding the thread objects. The respective `run()` methods of the thread objects will be executed as and when the threads are scheduled for execution.

8.8 MULTITHREADING—USING `isAlive()` AND `join()`

The main thread should always be the last thread to end, i.e., all the child threads spawned out of the main thread should end executing before the main itself. The execution of main thread can be prolonged using `Thread.sleep()` method. The time for which the main should be made to sleep cannot be estimated exactly, as the time taken by child thread is difficult to estimate. If we fall short on our estimation, then the child thread will terminate after the termination of main thread, which is not something we want. This is the lacuna associated with sleep technique.

Two other methods which generally work in tandem can be used to resolve the above mentioned crisis: `isAlive()` and `join()`. Both the methods as mentioned in Table 8.1 are part of the `Thread` class. `isAlive()` method returns a boolean value. It returns 'true' if the thread is active, i.e., it has started and not stopped. If it returns 'false', the thread can either be a new thread or a dead thread. In other words, if `isAlive()` returns 'true', the thread can be comprehended to be either in 'Runnable' or 'Not Runnable' state, otherwise it is in 'Dead' state. The `isAlive()` method can be used to check whether the child thread is running or not. The method as defined in the `Thread` class is as follows:

```
final boolean isAlive()
```

As far as the `join()` method is concerned, it waits until the thread on which it is called terminates. It waits for the child thread to terminate and then *joins* the main thread. Apart from this, `join()` method can also be used to specify the amount of time you want the child thread to wait before terminating. Example 8.6 shows the use of these two methods.

Example 8.6 Use of `join()` in Forcing a Thread to Wait for Other's Termination

```
L1  class ThreadJoin implements Runnable {
L2  String thread;
L3  Thread thrd;
```

```
L4   ThreadJoin (String threadName) {
L5   thread = threadName;
L6   thrd = new Thread (this, thread);
L7   thrd.start();
L8   }
L9   public void run() {
L10  try {
L11  Thread.sleep(2000);
L12  for(int i = 1; i<= 3; i++) {
L13  System.out.println("\t From child thread " + thread + " : i = "+i);
L14  }
L15  } catch(InterruptedException e ) {
L16  System.out.println("Exception: Thread "+ thread + " interrupted");
L17  }
L18  System.out.println("Terminating thread: " + thread );
L19  }
L20  }
L21  class JoinDemo {
L22  public static void main (String args []) {
L23  ThreadJoin threadA = new ThreadJoin ("A");
L24  ThreadJoin threadB = new ThreadJoin ("B");
L25  ThreadJoin threadC = new ThreadJoin ("C");
L26  ThreadJoin threadD = new ThreadJoin ("D");
L27  System.out.println("Thread Status: Alive");
L28  System.out.println("Thread A: " +threadA.thrd.isAlive());
L30  System.out.println("Thread B: " +threadB.thrd.isAlive());
L31  System.out.println("Thread C: " +threadC.thrd.isAlive());
L32  System.out.println("Thread D: " +threadD.thrd.isAlive());
L33  try {
L34  System.out.println("Threads Joining......");
L35  threadA.thrd.join();
L36  threadB.thrd.join();
L37  threadC.thrd.join();
L38  threadD.thrd.join();
L39  } catch (InterruptedException e){
L40  System.out.println("Exception: Thread main interrupted.");
L41  }
L42  System.out.println("Thread Status: Alive");
L43  System.out.println("Thread A: " + threadA.thrd.isAlive());
L44  System.out.println("Thread B: " + threadB.thrd.isAlive());
L45  System.out.println("Thread C: " + threadC.thrd.isAlive());
L46  System.out.println("Thread D: " + threadD.thrd.isAlive());
L47  System.out.println("Terminating thread: main thread.");
L48  }
L49  }
```

Output

```
Thread Status: Alive
Thread A: true
Thread B: true
Thread C: true
```

```
  Thread D: true
  Threads Joining.....
    From child thread A :i = 1
    From child thread A :i = 2
    From child thread A :i = 3
Terminating thread: A
    From child thread B :i = 1
    From child thread B :i = 2
    From child thread B :i = 3
Terminating thread: B
    From child thread C :i = 1
    From child thread C :i = 2
    From child thread C :i = 3
Terminating thread: C
    From child thread D :i = 1
    From child thread D :i = 2
    From child thread D :i = 3
Terminating thread: D
  Thread Status: Alive
  Thread A: false
  Thread B: false
  Thread C: false
  Thread D: false
  Terminating thread: main thread.
```

Explanation

(*Only those lines relevant to the topic are explained.*)

L28–32 After the threads are declared using the constructor of the `MyThread` class, the `isAlive()` method is called for each thread. The value returned by the `isAlive()` method is then displayed on the screen.

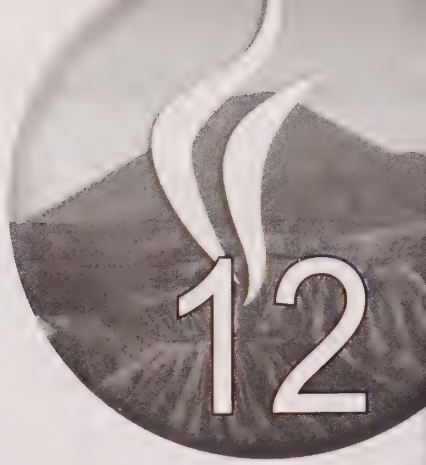
L35–38 The `join()`method is called for each thread. The `join()` method causes the main thread to wait for all child threads to complete execution before the main thread terminates.

L43–46 Again the `isAlive()` method is used with each thread's object to check whether the child threads are alive or dead and the boolean values are displayed on the screen.

8.9 SYNCHRONIZATION

There can be instances when two or more threads access a common resource, say a common data or file. In order to maintain consistency, it becomes imperative that the resource is made available to only one thread at a time. For example, there are two threads, one responsible for writing to a file (here, a resource) and other for reading from the same file. If both the threads start concurrently, both would try to access the file at the same time. Obviously, if the first thread has not written the values completely, the values read by the second thread would be inconsistent. Java has such inbuilt mechanism, which lets only one thread use a resource at a time, known as *synchronization*. Usually, operating systems do provide for such mechanism, but Java has a unique language level support for it.

How does it work? Many of you, having the knowledge of operating systems, would know what *semaphores* are. Likewise, we have the concept of *monitors* here. *Monitor* is an object that



Applets

Ideologies, however appealing, cannot shape the whole structure of perceptions and conduct unless they are embedded in daily experiences that confirm them.

Christopher Lasch

After reading this chapter, the readers will be able to

- understand the difference between applet and application
- understand the lifecycle of an applet
- learn how applets are created and executed
- create GUI within applets

12.1 INTRODUCTION

Many of you must have come across the word 'Applet'. What is this applet? Applets are basically small Java programs which can be easily transported over the network from one computer to other. This is the reason why applets are used in Internet applications, where these applets (i.e., small Java programs), embedded in an html page, can be downloaded from the server and run on the client, so as to do a specific kind of job. These applets have the capability of displaying graphics, playing sound, creating animation, and performing other jobs that can be done by simple application programs. To execute these applets on the client, the client must have either a Java-enabled browser or a utility known as appletviewer (comes as part of JDK).

If we could develop simple Java standalone application programs, what was the need to have applets? Actually applets are not full-featured programs, as these are usually written to accomplish small tasks or part of bigger tasks. Before getting further, you must understand the difference between an applet and an application. Table 12.1 illustrates the differences between applets and applications.

In Java, applets can be dealt in two ways. One is the conventional applets, which are directly evolved from 'Applet' class. Theses applets use Abstract Window Toolkit (AWT) to get the GUI features. The other kinds of applets are those which are based on swing class, `JApplet`. We will discuss AWT-based applets in this chapter, while swing-based applets will be discussed in Chapter 15.

Table 12.1 Difference between Applet and Application

Applet	Application
The execution of the applet does not start from `main()` method, as it does not have one.	The execution of an application program starts from `main()`.
Applets cannot run on their own. They have to be embedded inside a web page to get executed.	These can run on their own. In order to get executed, they need not be embedded inside any web page.
Applets can only be executed inside a browser or appletviewer.	Applications are executed at command line.
Applets execute under strict security limitations that disallow certain operations (sandbox model security).	Applications have no inherent security restrictions.
Applets have their own life cycle `init()`→`start()`→`paint()`→`stop`→`destroy()`	Applications have their own life cycle. Their execution begins at `main()`.

12.2 APPLETS

`java.applet.Applet` is the superclass of all the applets. Thus all the applets, directly or indirectly, inherently use the methods of `Applet` belonging `java.applet` package. This class provides all the necessary methods for starting, stopping, and manipulating applets. It also has methods providing multimedia support to an applet. `Applet` class has a predefined hierarchy in Java, which shows the classes extended by `Applet` class.

Figure 12.1 simply makes it easy for you to understand that an applet, which is a subclass of `java.applet.Applet`, also inherits the methods of the other classes like `java.awt.Panel`, `java.awt.Container`, `java.awt.Component`, and `java.lang.Object`, indirectly.

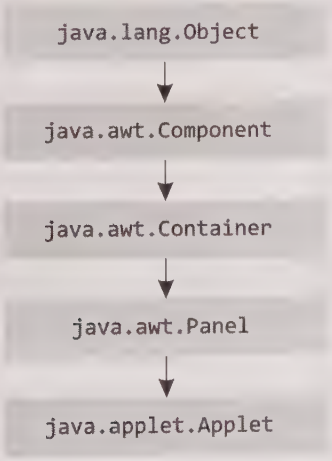


Fig. 12.1 Hierarchy of `Applet` Class

You can see that these classes are the ones which provide support for Java's window-based GUI, thus making an applet capable of supporting window-based activities. The common methods belonging to the `Applet` class are mentioned in Table 12.2.

Table 12.2 `Applet` Class Methods

Method	Description
`void init()`	First method to be called when an applet begins execution.
`boolean isActive()`	Returns true if the applet is running, otherwise false.
`URL getDocumentBase()`	Gets the URL of the document in which this applet is embedded.
`URL getCodeBase()`	Returns the URL of the directory where the class file of the invoking applet exists.
`String getParameter (String name)`	Returns the value of the parameter associated with parameter's name. Null is returned if the parameter is not specified.
`AppletContext getAppletContext()`	Determines this applet's context, which allows the applet to query and affect the environment in which it runs.
`void resize (int width,int height)`	Resizes the applet according to the parameters, *width* and *height*.
`void showStatus(String msg)`	Displays the string, *msg*, in the status window of the browser or appletviewer (only if they support status window).
`Image getImage(URL url)`	Returns an object of image, which binds the image found at the URL, specified as the argument of the method.
`Image getImage(URL url, String imgName)`	Returns the image object which encapsulates the image found at the specified URL and having the name specified by imgName.
`static final AudioClip newAudioClip(URL url)`	Returns an `AudioClip` object that encapsulates the audio found at the URL specified as the argument.
`void start()`	Starts or resumes the execution of applet.
`void stop()`	Stop or suspends the applet.
`void destroy()`	Terminates the applet.
`AccessibleContext getAccessibleContext()`	Returns the accessibility context for the invoking object.
`AudioClip getAudioClip(URL url)`	Returns the `AudioClip` object, which encapsulates the audio clip found at the URL, specified as the argument to the method.
`AudioClip getAudioClip(URL url,String clipName)`	Returns the `AudioClip` object, which encapsulates the audio clip found at URL, specified as the argument to the method and having the name specified by `clipName`.
`String getAppletInfo()`	Returns the string describing the applet.
`Locale getLocale()`	Returns the `Locale` object that is used by various locale sensitive classes and methods.
`String[][] getParameterInfo()`	Returns a string table that describes the parameter recognized by the applet.

12.3 APPLET STRUCTURE

Apart from using the services of `Applet` class, an applet also uses the services of `Graphics` class of the `java.awt` package. The `Applet` class has methods such as `init()`, `start()`, `destroy()`, and `stop()`, which are responsible for the birth and behavior of an applet. We have already mentioned

in Section 12.1 that unlike an application program, Java runtime system does not call the `main()` method to start the execution of an applet, rather it just loads the methods of `applet` class which are responsible for starting, running, stopping, and manipulating an applet. The complete life cycle of the applet will be taken up in the next section.

There is a method, `paint()` in the `Container` class, which is inherited by `Applet` class (as you can make out from Fig. 12.1), carrying the signature,

```
public void paint(Graphics g)
```

This method, when called, displays the output of applet as per the code written on the applet's panel. You can see the argument of this method; it is nothing but an object of `Graphics` class. The object makes it possible for an applet to output text, graphics, sound, etc. One thing you must remember, you cannot take the services of `Graphics` class unless you import the package it belongs to, i.e., `java.awt`. The output operations for an applet requires the methods contained in the `Graphics` class, that is why its `Graphics` object is passed as argument to `paint()`. Now that you know some details about the internals of an applet, we can discuss the program structure of an applet.

When an applet is first loaded, Java runtime system creates an instance of the main class, which is `FirstApplet` in this case. Then the methods belonging to the `Applet` class are called through this object.

12.4 AN EXAMPLE APPLET PROGRAM

Let us take an example applet, which displays the statement "This is my first applet program."

Applet Program Structure

```
import java.awt.*;
//so as to make Graphics class available

import java.applet.*;
//so as to make Applet class available
.....
.....

public class NewApplet extends Applet
// new applet with the name, newApplet declared
{
     .....
     .....

     public void paint(Graphics g)
     // paint() of Applet class overridden to contain output operations
     {
     .....
     .....
     }
.....
.....
}
```

Example 12.1(a) **First Applet Example**

```
L1   import java.applet.*;
L2   import java.awt.*;
L3   public class FirstApplet extends Applet
     {
L4        public void paint(Graphics g) {
L5        g.drawString("This is my First Applet", 10, 10);
L6        }
L7   }
```

Explanation

L1 All applets are the subclasses of `Applet` class. All applets must import the `java.applet` package.
L2 The applet uses the methods of `java.awt` package; it must be imported.
L3 The `FirstApplet` class is declared public so that the program that executes the applet (a Java-enabled browser or applet viewer, which might not be local to the program) can access it. This class extends the `Applet` class of `java.applet` package, thus inheriting the features of `Applet` class.
L4 The `paint()` method defined by AWT `Container` class is overridden. Any output to be shown by an applet has to be taken care by this method only. Please note that an object of `Graphics` class is passed as parameter to this method.
L5 The object of the `Graphics` class is used to invoke `drawString()` method, which is responsible for printing the string ("This is my First Applet") at x-coordinate 10 and y-coordinate 10.

12.4.1 How to Run an Applet?

There are two ways to run an applet. We will explain these in context to the above example. These approaches are

(a) Save the file as `FirstApplet.java` and compile it by using `javac`. Now, type in the following HTML code in your editor and save the file as `FirstApplet.html` (here, the file name is not necessarily the same as the class name, as it was for the `java` file.)

```
<HTML><BODY>
<APPLET code = "FirstApplet.class" WIDTH = 200 HEIGHT =
150></APPLET>
</BODY></HTML>
```

You can execute the HTML file by giving

```
appletviewer FirstApplet.html
```

Note If you wish to run the above html file in any web browser, instead of using applet viewer, you must have Java-enabled web browser. Otherwise, you will have to install Java plug-in, which lets you run your applets as web pages under 1.2 version of JVM instead of the web browser's default virtual machine.

(b) Just as above, save the file as `FirstApplet.java` and compile it by using `javac`. In order to run the applet, you have to give the below HTML coding as a comment in `FirstApplet.java`.

```
/* <APPLET code = "FirstApplet.class" WIDTH = 200 HEIGHT = 150></APPLET> */
```

Execute the applet as,

```
appletviewer FirstApplet.java
```

In this chapter, we will be using the second approach throughout. So Example 12.1(a) should have been actually written as shown in Example 12.1(b).

Example 12.1 (b) First Applet Example (Revised)

```
/* <APPLET code = "FirstApplet.class" WIDTH = 200 HEIGHT = 150></APPLET>
*/ import java.applet.*;
import java.awt.*;
public class FirstApplet extends Applet {
   public void paint(Graphics g) {
      g.drawString("This is my First Applet",10,10);
}}
```

Output

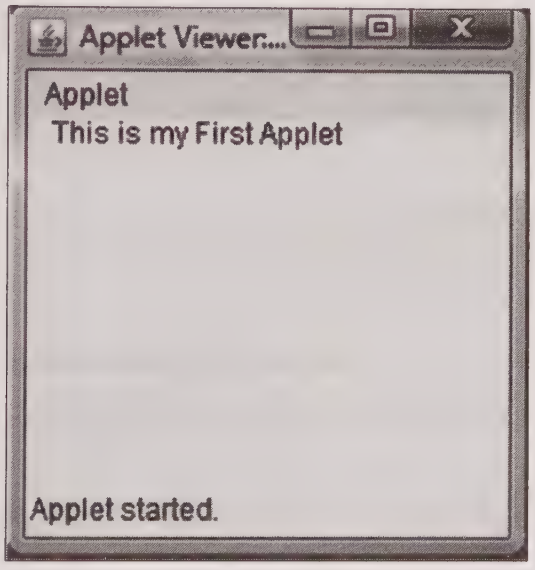


Fig. 12.2 Output Shown with the Help of Applet Viewer

12.5 APPLET LIFE CYCLE

An applet may move from one state to another depending upon a set of default behaviors inherited in the form of methods from `Applet` class. These states can be summed up as,

- Born
- Running

- Idle
- Dead

Figure 12.3 shows the flow an applet takes while moving from one state to another.

As mentioned before, an applet may override some of the basic methods of class `Applet`. Note that these methods are responsible for the lifecycle of an applet. These methods are

- `init()`
- `start()`
- `stop()`
- `destroy()`

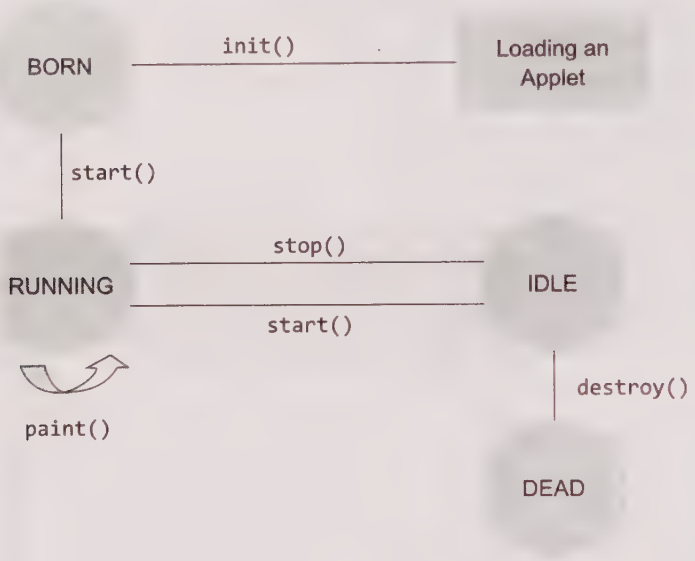


Fig. 12.3 Applet's State Diagram

Let us discuss all these states of an applet in greater detail.

Born State

You can easily see from Fig. 12.3, that an applet enters this phase as soon as it is first loaded by Java. This is made possible by calling `init()` of `Applet` class. Now, what are the things that Java runtime system does, while loading the applet, i.e., when `init()` is called? It creates the objects needed by the applet; it might set initial values, load font and images or set up colors. The method `init()` is called only once during the lifetime of an applet.

Note In order to initialize an applet, we must override the `init()` method of `Applet` class.

Running State

Applet moves to the running state by calling `start()`. An applet moves to this phase automatically after the initialization state. But if the applet is stopped or it goes to idle state, `start()` must be called in order to force the applet again to the running state. Suppose you have opened a web

page (having an applet) and you move temporarily to another web page (by minimizing it) the first one goes to the idle state; when you return back to the first page, `start()` is called to put the applet in the running state again. Unlike `init()`, `start()` can be called more than once.

Note `start()` can be overridden to create a thread to control an applet.

You can see the `paint()` method in Fig. 12.3. This method is responsible for forcing the applet to an intermediary state (display state), which is actually a part of the running state itself. While running, an applet may need to perform some output and display it on the panel of the applet. The `paint()` method, which is a part of `Container` class (a superclass of `Applet` class), needs to be overridden for the purpose. This method is called each time to draw and redraw the output of an applet. We already know the drawing of output of an applet. Let us discuss redrawing the output of an applet with an example. An applet window may be minimized and then restored. This restoration is nothing but redrawing of applet's output and could be achieved by calling `paint()`. Actually when an applet in restored, `start()` and `paint()` are called in sequence. We will revisit this method in Section 12.7.1.

Idle State

An applet goes to idle state, once it is stopped from running. If we leave a web page containing an applet (i.e., minimize it), the applet automatically goes to idle state. An applet can also be forced to stop or go to idle state by calling `stop()`.

Note If a thread has been created to control an applet by overriding `start()`, then we must use `stop()` to stop the thread, by overriding the `stop()` method of the `Applet` class.

Dead State

Terminating or stopping an applet should not be confused with destroying an applet. An applet goes to dead state when it is destroyed by invoking the `destroy()` method of `Applet` class. It results in complete removal of applet from the memory. Whenever we quit the browser, `destroy()` is called automatically. You should free up the resources being used by applet (if any) by overriding the `destroy()` method. Like `init()`, `destroy()` is also called only once. `stop()` is always called before `destroy()`.

12.6 COMMON METHODS USED IN DISPLAYING THE OUTPUT

There are certain methods which you should be acquainted with, as they might be used in applet programming further down the chapter. These are the methods belonging to different classes, which can handle the AWT windowed environment.

`drawString()`

This method is a member of `Graphics` class, used to output a string to an applet. It is typically called from within the `paint()` or `update()` method. Its form is

```
void drawString(String msg, int a, int b)
```

Here, the `string msg` is the string output to be displayed by the applet and `a, b` are the `x, y` coordinates respectively of the window, where the output has to be displayed.

setBackground()

This method belongs to component class. It is used to set the background color of the applet window. Its form is

```
void setBackground(Color anyColor)
```

The above method takes the color to be set as background, as argument. The `Color` class has certain predefined constants for each color, such as `Color.red, Color.blue, Color. green,` and `Color.pink`.

setForeground()

This method is similar to `setBackground` method, except that these are used to set the color of the text to be displayed on the foreground of the applet window. Its form is

```
void setForeground(Color anyColor)
```

Component class has two more methods `getBackground()` and `getForeground()`, having the following forms:

```
Color getBackground();
Color getForeground();
```

You can very well see that these methods return the current context of the `Color`, showing the background and foreground colors, respectively.

showStatus()

This method is a member of `Applet` class. It is used to display any string in the status window of the browser or `appletviewer`. Its from is

```
void showStatus(String text)
```

Here, the argument of the method is basically the string which you want to be displayed in the status window.

Before going any further, we should better take an example which uses these methods, discussed until now.

Example 12.2 Applet Methods

```
/* <APPLET code = "ExampleApplet.class" WIDTH = 200 HEIGHT = 150></APPLET> */
L1   import java.applet.Applet;
L2   import java.awt.Color;
L3   import java.awt.Graphics;
L4   public class ExampleApplet extends Applet{
L5   String text;
L6   public void init() {
L7      setBackground(Color.white);
L8      setForeground(Color.red);
L9      text = "This is an example applet";
L10  System.out.println("....Initialized the applet"); }
```

```
L11    public void start() {
L12       System.out.println("....Starting of the applet");
L13    }
L14    public void stop() {
L15       System.out.println("....Stopping the applet");
L16    }
L17    public void destroy() {
L18       System.out.println("....Exiting the applet");
L19    }
L20    public void paint(Graphics g) {
L21       System.out.println("....Painting the applet");
L22       g.drawString(text, 30, 30);
L23       showStatus("This is status bar"); }}
```

Output

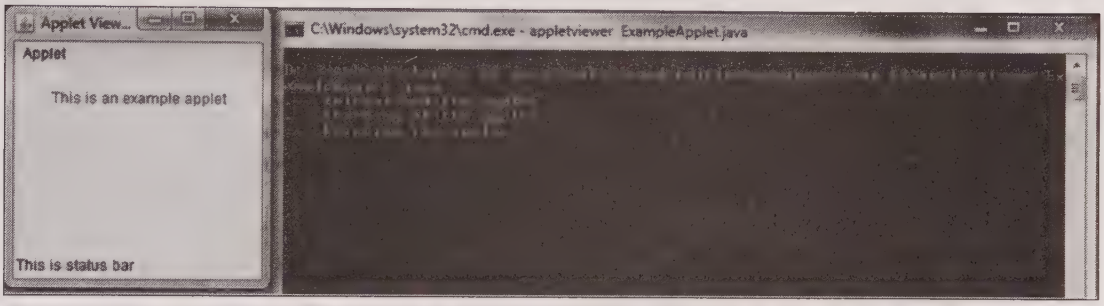


Fig. 12.4(a) Applet Initialized Using Applet Viewer

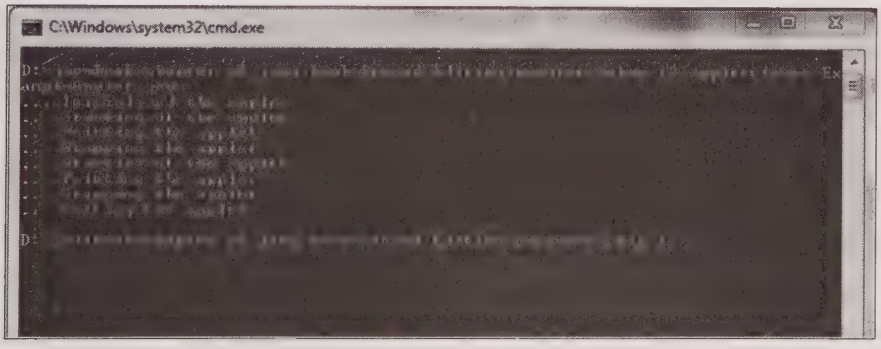


Fig. 12.4(b) Strings Printed by Various Methods of the Applet

Explanation

L1–3 All the important classes (belonging to their respective packages), whose members are to be used in the applet are imported.

L6–10 This section shows the implementation of `init()`, where the background and foreground of the applet is set to white and red, respectively (see L7–8). White and red are static fields of the `Color` class (part of `java.awt` package). In L9, the text is initialized by a string, `This is an example applet`.

L11–13 These lines account for the implementation of `start()`, responsible for forcing the applet in running state. L13 displays the message about the start of the applet on the screen. Note that this message will not be displayed on the applet window;

it will be displayed as seen by you in earlier chapters (by the use of `System.out.println()`).

L14–16 These lines take care of the implementation of `stop()`. L16 just displays the message about stopping an applet. As many a times you will stop the applet, this message will be displayed on the screen. You can visualize easily that even minimizing the applet window stops or forces the applet into idle state. If restored after getting minimized, it will again invoke `start()` and `paint()`.

L17–19 These lines take care of the implementation of `destroy()`. Try closing the applet window and you will see the message ".....Exiting the applet" (L19). Here, this message simply means that the applet is destroyed or has moved to the dead state.

L20–23 These lines are accountable for the implementation of `paint()` method. In L22, `Graphics` object `g` is used to invoke its `drawString()` method, which is actually used for writing on an applet window. See the arguments passed to this method: *text*, which contains the string, "This is an example applet" and the *x*, *y* coordinates from where this text will start in the displayable part of the applet. `paint()` is also called when the window containing applet is covered by another window and they later uncovered. (not minimized and restored)

L22 You can see the method, `showStatus()`, having the text, which has to be shown in the status window of the applet, as argument.

12.7 `paint()`, `update()`, and `repaint()`

All components and containers (since containers are actually components) in the JDK have two methods that are called by the system to paint their surface. These methods are `paint()` and `update()`, belonging to component class (see Fig. 12.1). The signature of these methods are shown below,

```
public void paint(Graphics g);
public void update(Graphics g);
```

If you wish that a drawing should appear in a window, you shall override either or both of the methods. Let us discuss these methods in detail.

12.7.1 `paint()` Method

When a component needs to draw/redraw itself, its `paint()` method is called. The component draws itself when it first becomes visible. The component `paint()` method is also invoked when the window containing it is uncovered, if it is covered by another window.

The simplest `paint()` method looks like the following:

```
public void paint(Graphics g) {... }
```

We have discussed the `Graphics` object passed to the method earlier. It will be discussed in more detail in the next chapter.

Example 12.3 Set the Color of the Applet and Draws a Fill Oval

```
/* <APPLET code = "FillOval.class" WIDTH = 200 HEIGHT = 200></APPLET> */
L1   import java.applet.Applet;
L2   import java.awt.Color;
L3   import java.awt.Graphics;
L4   public class FillOval extends Applet
     {
L5   public void paint(Graphics g)
     {
```

```
L6          g.setColor(Color.red);
L7          g.fillOval(20, 20, 60, 60);
L8          }
L9      }
```

Output

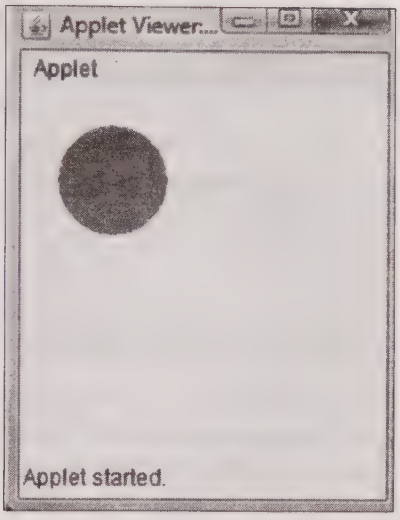


Fig. 12.5 Fill Oval with Red Color

Explanation

L5–8 These lines are accountable for the implementation of `paint()` method. The `setColor()` method of `Graphics` class is used to set the drawing color of the applet to red (L6). Another method, `fillOval()`, belonging to the `Graphics` class is invoked at L7. It fills an oval bounded by the specified rectangle with the current color. The parameters passed to the method are the x-coordinate, y-coordinate, width, and height, respectively.

12.7.2 `update()` Method

Another method which does the same job as `paint()` method, which is called by AWT components to paint its surface is the `update()` method.

Now let us discuss what this method does. It clears the surface of the calling component to its background color and then calls `paint()` to paint the rest of the component. It makes the job easier because one does not have to draw the whole component within a `paint()` method, as the background is already filled. Then, when one overrides `paint()`, he/she only needs to draw what should appear on the foreground.

It simply does not mean that you will never override `update()`. Let us consider a case where you would like to draw a large, red oval inside a window having yellow background. Now take the case where you do not override `update()`, the window's entire background will be drawn by

the `update()` method, and then the red oval will be drawn by the `paint()` method. A large area of one color is first drawn and then the large area of the oval in another color (red in this case) is redrawn. You, as a user can see some slight flickering while displaying the result, especially if you try to draw the oval a number of times in succession.

The above problem can be overcome by overriding `update()`. You would override `update()` to call `paint()`. Then this `paint()`will first draw only the background areas surrounding the red oval and then draw the red oval. Obviously, the flickering problem found earlier is removed because of elimination of the drawing of two overlapping objects of different colors.

12.7.3 `repaint()` Method

Sometimes you may want to force a component to be repainted manually. For example, if you have changed certain properties of a component to reflect its new appearance, you can call the `repaint()` method. Here is an example:

```
text.setBackground(Color.blue);
text.repaint();
```

Calling the `repaint()` method causes the whole component to be repainted.

```
repaint() → update() → paint()
```

`repaint()` in its default implementation calls `update()` which in turn calls `paint()`. `repaint()` method requests the AWT to call `update` and it returns. The AWT combines multiple rapid `repaint` requests into one request (usually this happens when you `repaint` inside a loop). So the last repaint in the sequence actually causes `paint()`. We will discuss these topics in Chapters 13 and 14, where we will illustrate the use of `repaint()` with proper examples.

12.8 MORE ABOUT APPLET TAG

We have used the APPLET tag while writing code for applets. An applet has to be specified in an HTML file. This is done by using APPLET tag in an HTML file. Applets are executed by a Java-enabled web browser as soon as it encounters the APPLET tag inside the HTML file. If you want to view and test an applet using the utility, `appletviewer`, of JDK, you just have to include a comment containing the APPLET tag, just above the actual applet code. We have already discussed how that has to be done.

Till now, we have been using the following form of APPLET tag:

```
<APPLET CODE = filename WIDTH = pixels HEIGHT = pixels></APPLET>
```

This is the most simplified form of APPLET tag, having only the mandatory fields as attributes. Actually this particular tag has many more attributes which are optional but worth discussing. The full syntax of the APPLET tag is shown below.

```
<APPLET [CODEBASE= codebasedURL]
CODE = appletFile [ALT= alternateText] [NAME = appletInstanceName] WIDTH
= pix els HEIGHT = pixels [ALIGN = alignment] [VSPACE = pixels]
[HSPACE = pixels]>
[<PARAM NAME = attributeName VALUE = attributeValue>]
[<PARAM NAME = attributeName VALUE = attributeValue>]
.....
</APPLET>
```

In the above syntax, the attributes which are put inside the big braces are optional ones. Let us discuss about the use of these attributes in detail.

Codebase Here, we may specify the URL of the directory where the executable class file (specified by CODE attribute) of the applet will be searched for.

Code It gives the name of the file containing the applet's compiled class file. It is a mandatory attribute, which should always be present in APPLET tag.

Alt It is an attribute, which is used to specify the alternate short text message that should be displayed in case the browser recognizes the HTML tag but cannot actually run the applet because of some reason.

Name It is possible to give a name to an applet's instance using this optional attribute. If any other applet on the same web page wants to communicate with this applet, it is referenced through its NAME only.

Width It gives the width of the applet display area in terms of pixels.

Height It gives the height of the applet display area in terms of pixels.

Align This optional attribute is used to set the alignment of an applet. The alignment can be set as LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

Vspace These are used to specify the space, in pixels, above and below the applet.

Hspace These are used to specify the space, in pixels, on each side of the applet.

You can use PARAM tags between the <APPLET> and </APPLET> tags to provide information about parameters, or arguments, to be used by the Java applet. The <PARAM> tag is simple—it NAMES a parameter the JAVA applet needs to run, and provides a VALUE for that parameter.

Note User-defined parameters can be supplied to an applet using <PARAM.....> tags.

This tag has two parameters: NAME and VALUE.
Name Attribute name.
Value Value of the attribute named by corresponding PARAM NAME.
The applets access their attributes using the `getParameter()` method. Its signature is as follows:

```
String getParameter(String name);
```

Let us take an example applet which uses the concept of passing parameters.

Example 12.4 **Param Tag**

```
/*<APPLET CODE = ParamPassing.class WIDTH = 300 HEIGHT = 250>
<param NAME = yourName VALUE = John>
<param NAME = yourProfession VALUE = consultant>
<param NAME = yourAge VALUE = 35>
</applet>*/

L1    import java.awt.*;
L2    import java.applet.*;
```

```
L3    public class ParamPassing extends Applet {
L4    String name;
L5    String profession;
L6    int age;
L7    public void start() {
L8          String str;
L9          name = getParameter("yourName");
L10   if (name == null) name = "not found";
L11   str = getParameter("yourProfession");
L12   if (str != null) profession = str;
L13   else profession = "No job";
L14   str = getParameter("yourAge");
L15   try {
L16   if (str != null) age = Integer.parseInt(str);
L17   else age = 0;
L18 } catch (NumberFormatException e) {}
L19 }
L20 public void paint(Graphics g) {
L21 g.drawString("your name: "+name, 10, 10);
L22 g.drawString("your profession: "+profession, 10, 30);
L23 g.drawString("your age: " +age, 10, 50);
L24 }}
```

Output

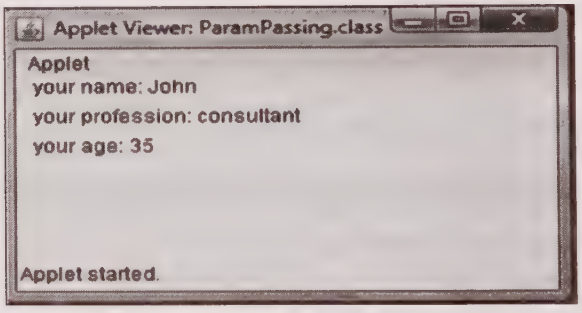


Fig. 12.6 Parameter Passing through Applets

Explanation

Let us start with the APPLET tag placed as comment before the actual code for the program starts. Three PARAM tags are between the start and close of the APPLET tag. All the three PARAM tags have NAME and its corresponding VALUE, such as *yourName* has the value *John*, *yourProfession* has the value *consultant* and *yourAge* has the value 35.

L1–2 For importing necessary classes from their respective packages.

L3 A class named as `ParamPassing` is declared to extend the `Applet` class of the `java.applet` package.

L4–6 References to the `String` class is declared as `name` and `profession` in L4 and L5, respectively. A variable, `age`, of integer type is declared in L6.

L7–19 Implementation of `start()` method is shown. At L9, `getParameter()` method returns the value of the parameter name, `yourName`, passed as argument. The returned value is stored in `name`, declared at L4 as string reference. Similarly, two more `getParameter()` methods are used at L11 and L14, returning the values for the respective parameters names passed as arguments. The use of

these values returned by `getParameter()` methods is too simple to be explained here. Only point worth mentioning here is the use of wrapper class, `Integer`, at L16. The value returned by `getParameter()` at L14 is of `String`type, and it needs to be converted to integer, which is done by the use of `parseInt()` method of `Integer` class. This method throws an exception, named as `NumberFormatException`, which is caught at L18.

L20–24 Implementation of `paint()` method is shown, where values for name, profession, and age are displayed on the applet window.

Note The APPLET parameters stored in the PARAM tag actually have little directly to do with HTML. As you have already seen in the previous example, that it is the responsibility of the applet to check the parameter values and respond accordingly. One more interesting thing worth noting here is that you can increase the flexibility of the applet by making the applet work in multiple situations without recoding and recompiling it, by defining and redefining the parameters (as these are placed as comments, so they are not checked during compilation).

We can sum up the use of PARAM tag for passing parameters to applets in two steps:

- Place the PARAM tag with names and corresponding values between the start and end of the APPLET tag.
- Write the needed code for the applet to retrieve these parameter values, as shown in Example 12.4.

12.9 `getDocumentBase()` AND `getCodeBase()` METHODS

Suppose a directory holds the HTML file, responsible for starting the applet and you need the applet to load data (i.e., media and text) from this directory, which is known as document base. We can get the URL of this directory in the form of URL object by using the method `getDocumentBase()`. Similarly, there is another method, `getCodeBase()`, which returns the URL object of the directory from where the class file of the applet is loaded. The following example illustrates the use of these methods:

Example 12.5 **`getCodeBase()` and `getDocumentBase()` Methods**

```
/*<APPLET CODE = BaseMethods.class WIDTH = 300 HEIGHT = 250></applet>*/
L1   import java.awt.*;
L2   import java.applet.*;
L3   import java.net.*;
L4   public class BaseMethods extends Applet {
L5   public void paint(Graphics g) {
L6   String str;
L7   URL url;
L8   url = getCodeBase();
L9   str = "Code Base: "+url.toString();
L10  g.drawString(str, 20, 40);
L11  url = getDocumentBase();
L12  str = "Document Base: " +url.toString();
```

Explanation

L1–2 Imports the `awt` and the `applet` packages.
L3 Commented html APPLET tag meant to run the applet through the applet viewer utility.
L4 Public class `MediaTrackerDemo` is declared to inherit the `Applet` class.
L5–8 Instance variables are created of type Image array, `MediaTracker`, `Thread` and an `int` variable meant to change the images. Image array is defined to store images. `MediaTracker` instance will be tracking the status of images stored in the image array. A `Thread` instance is required for changing the images and it acts as a counter for deciding which image to be shown.
L9–15 `init` method is defined for the applet. So all one-time initializations will be done in this method. L10 creates a `MediaTracker` object and `this` is passed as an argument within the constructor to signify the component on which the images will be drawn. L11 creates an image array of two images. The image array is populated with two images in L12 and L14 using the `getImage` method. We have already discussed this method in the above example. These images are added to the `MediaTracker` instance using the `addImage` method in L13 and L15. The first argument is the image and the second is an integer id which is used to track the image.
L16–17 Instantiates the thread and starts it. Remember `this` (current object) is passed in the constructor of `Thread` class to signify the object (i.e., the applet in our case) whose `run` method will be invoked when this thread is started.
L18–21 `paint` method is declared. L19 set the color as white. L20 fills the rectangle (i.e., entire applet) with white `color` so it erases whatever is there on the applet in every paint attempt and then restores the black color in L21. (It is better to whitewash the applet so that there is no overlapping of images).
L22 Checks whether all images have loaded and if yes, draws the images on the applet using the `drawImage` method in L23 else displays a string mentioned in L27. We have already discussed the `drawImage` method above.
L24–25 Shows statements to change the images. We are using the current variable as an index into the image array which is incremented to change the image. The value of the current variable is checked against the length of the image array (so that all images in the image array are shown) and then reset to 0.
L28 `run` method is defined to tell what the thread is supposed to do.
L29 `try` block defined to catch the exceptions that arise in the code.
L30–33 `waitForAll()` method has been used on the tracker object to wait for all images to finish loading and then an infinite loop is executed (L31) to repaint (L32) the applet. Thread is made to sleep in L33 to introduce a delay effect.
L34 Catches the exception raised by sleep method, if any.

12.13 `Graphics` CLASS

You know about applets in Java now. But these applets are incomplete without their ability to draw graphics. You can write Java applets that can draw lines, figures of different shapes, images, and text in different fonts, styles, and colors. Every applet has its own area on the screen known as canvas, which is actually the display area. To create this display area, you must have the knowledge of Java coordinate system. This coordinate system has the origin (0, 0) in the upper-left corner. Positive x values are to the right and positive y values to the bottom. The values of (x, y) are in pixels.

Now let us discuss the class which makes the use of graphics possible in Java. It is the `Graphics` class belonging to the `java.awt` package, defined as

```
public abstract class Graphics extends Object
```

Note The `Graphics` class is the abstract base class for all graphics contexts that allows an application to draw onto components that are realized on various devices, as well as onto off-screen images.

This class has a number of methods defined in it, some of which are mentioned below in Table 12.5.

Table 12.5 Few Methods of the `Graphics` Class

Name	Description
`drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	Draws the outline of a circular or elliptical arc covering the specified rectangle.
`drawLine(int x1, int y1, int x2, int y2)`	Draws a line.
`finalize()`	Disposes of this graphics context once it is not referenced.
`translate(int x, int y)`	Translates the origin of the graphics context to the point (*x*, y) in the current coordinate system.
`drawOval(int x, int y, int width, int height)`	Draws the oval.
`drawPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Draws a polygon defined by arrays of *x* and *y* coordinates.
`drawRect(int x, int y, int width, int height)`	Draws the specified rectangle.
`getClip()`	Returns the bounding rectangle of the current clipping area.
`fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	Fills a circular or elliptical arc covering the specified rectangle.
`fillOval(int x, int y, int width, int height)`	Fills an oval bounded by the specified rectangle with the current color.
`fillPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Fills a closed polygon defined by arrays of *x* and *y* coordinates.
`fillRect(int x, int y, int width, int height)`	Fills the rectangle.
`getColor()`	Returns this graphic context's current color.
`getFont()`	Returns the current font.
`setColor(Color c)`	Sets the drawing color.
`setFont(Font font)`	Sets this graphic context's font to the specified font.
`fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	Fills the specified rounded corner rectangle with the current color.
`drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	Draws an outlined round-cornered rectangle using this graphic context's current color.
`drawString(String str, int x, int y)`	Draws the string on the specified coordinates.
`clearRect(int x, int y, int width, int height)`	Clears the specified rectangle by filling it with the background color of the current drawing surface.

(*Contd*)