06 Multithreading and Packages in Java

IN THIS CHAPTER

Thread is the smallest unit of an executable code in a program. It helps you divide a process in multiple parts to speed up the process. A process is a program that executes as a single thread.

Programs that run as single threads can cause problems when you perform two or more tasks simultaneously. For example, while playing an online game, you sometimes come across a situation where the game does not show the updated score but still displays the graphics properly. Here, displaying the graphics of the game and updating the score are two different tasks that are to be handled simultaneously. Therefore, it is important to ensure that the game provides support to handle these multiple tasks to make the game fast and interactive enough to play.

To overcome such problems, Java provides built-in support for multithreaded programming, which enables a single program to perform multiple tasks simultaneously. In multithreaded programming, each thread is assigned a task to perform, and each of the threads executes independently in a program. Multiple threads within a program share the same memory address space, and therefore share the same set of variables and objects among themselves.

This chapter discusses threads, their life cycle, and how they are created, run, and terminated. It also explains the concepts of thread synchronization, inter-thread communication, thread priorities, and deadlocks in threads. In addition, this chapter discusses an important concept, that is, the packages of the Java programming language. We begin by providing an overview of threads.

6.1 Overview of Threads

Threads are used to implement concurrency in Java programs. Concurrency is the simultaneous execution of multiple tasks. These tasks can be implemented either as separate processes, or as a set of threads within a single program. For example, working in a word processor while listening to an audio file on the same computer are two tasks that are

implemented as two separate processes. However, we often need a single program or process to perform these multiple tasks at the same time, such as a word processor that calculates the number of words in a document while printing the document. Here, the word processor is a single program that performs two tasks simultaneously, calculating the number of words in the document and printing the document. These two tasks are implemented as two separate threads within the word processor.

Threads are light-weight processes, because creating a new thread requires fewer resources than creating a new process. For example, a newly created thread shares the same address space with the other threads in a program, whereas every individual process has its own separate address space. Therefore, creating a thread involves less overhead than creating a process.

The simultaneous execution of threads is implemented with the help of time-slice. Time-slice is the time that each thread gets for its processing. Each thread executes according to the time slice allotted to it but to you it appears as if these threads were running at the same time.

The objective of multithreading is to use the idle time of a CPU. For example, the CPU remains idle until you enter a character from the keyboard or move the mouse.

Some important uses of threads are as follows:

- Threads are used in server-side programs to serve the needs of multiple clients on a network or the Internet. On the Internet, the server has to cater to the needs of thousands of clients at a time. Threads are used to handling multiple clients simultaneously.
- Threads are used to create games and animations. For example, you can use threads to show a picture in motion. In such a case, you have to set the thread in such a way that it sleeps for some specific period of time and then is activated for a specific period of time, and continues this cycle. In this way, threads can be used to create animations.
- In many games, a user generally has to perform more than one task simultaneously. Threads can be very useful in such situations.

2 Defining a Thread

Every application program has a control of execution that executes a set of instructions of a program. These instructions are executed in a predefined manner or path of execution. This path of execution is known as execution `thread`. Each program has at least one execution `thread`, although, depending on the requirement, more than one execution `thread` can be created in a program through which you can perform more than one task simultaneously.

In the `Java` programming language, a program can have more than one `thread`, where each thread has its own path of execution. All the `threads` in a program share the same address space, data, and code in a program; therefore, threads are considered as lightweight processes. `Threads` in `Java` are objects that have all the characteristics of an object, such as member data and methods. You can create a thread in the following two ways:

- By extending the Thread class
- By implementing the Runnable interface

You extend the Thread class and create a thread by instantiating the class. On the other hand, if you implement the Runnable interface to create a thread, then you need to pass the object of Runnable type in the Thread constructor. Whether or not you create any other `thread` in a

Java program, the main thread is always created. Let's discuss the main thread in the following section.

The Main Thread

During the execution of a program, the Java Virtual Machine (JVM) creates a user thread automatically to execute the main() method, which is called the main thread. If any other user thread is not created in the program, the main thread terminates once the main() method finishes its execution. Apart from the main thread, a Java program can also have child threads. Apart from the main thread, all the threads created in the program inherit the user thread status of the main thread. In this case, even if the main() method terminates, the program keeps running until the user threads finish their execution.

Let's now discuss how threads are created and used. A thread can be created in Java in the following ways:

- Extending the java.lang.Thread Class
- Implementing the java.lang.Runnable Interface

Let's discuss these ways in detail.

Extending the java.lang.Thread

Threads can be created in a Java program by extending the java.lang.Thread class. This is the simplest form of creating threads in a program. Some of the important facts to keep in mind while creating threads in this way are:

- The class that extends the Thread class overrides the run() method in which the code executed by the thread is defined
- The constructor of the Thread class may be called explicitly in the constructor of the class, extending it by using the super keyword to initialize the thread
- The class extending the Thread class invokes the start() method inherited from the Thread class to make the thread eligible for running

Consider the following code snippet that shows a class extending the Thread class to create a simple thread class:

```
class SampleThread extends Thread {
  public void run() {
      System.out.println("The code to perform specific function by thread");
  }
  .
  .
}
```

In the preceding code snippet, the SampleThread class is created by extending the Thread class. The run() method of the Thread class is overridden by the SampleThread class to provide the code to be executed by a thread. You can also overload the run() method in the program, as shown in the following code snippet:

```
class SampleThread extends Thread {
  public void run() {
      System.out.println("The code to perform specific function by thread");
```

```
    }
    public static void main(string ar[]) {
        public void run(string str) {
            System.out.println("This the run method overloaded inside main");
        }
    .
    .
    }
```

In the preceding code snippet, the run() method is overloaded with the run(String str) method. This method needs to be invoked explicitly as the `Thread` class expects the `run()` method without any argument. When a thread starts, the `run()` method is executed without any argument by the thread in a separate call stack. No new thread of execution is created as a result of the invocation of the overloaded `run()` method. Instead, the overloaded version of the `run()` method behaves like a normal method of a class.

In Java, you cannot extend a class from more than one class. Therefore, while creating a thread by extending the Thread class, you cannot simultaneously extend any other class. For example, while creating a Graphical User Interface (GUI) application by using the Swing API, you need to extend the JFrame class and, therefore, you cannot extend the Thread class in the same application to create threads. However, in such cases, you can use another way of creating threads, that is, by implementing the Runnable interface.

■ Implementing the java.lang.Runnable Interface

Threads can also be created by implementing the `Runnable` interface of the `java.lang` package. This package allows you to define a class that can implement the `Runnable` interface and extend any other class, if needed. Creating a thread by implementing the `Runnable` interface is very similar to creating a thread by extending the Thread class. The following code snippet shows how to create a thread by implementing the `Runnable` interface:

```
class SampleThread implements Runnable {
  public void run() {
      System.out.println("The code to perform specific function by thread");
  }
.
}
```

In the preceding code snippet, the `SampleThread` class is declared by implementing the `Runnable` interface and overriding the `run()` method in which it defines the code to be executed by a `thread`.

After discussing `threads` and the various ways of creating them, let's learn how a thread is instantiated so that it can be used in a program.

.3 Instantiating a Thread

A `thread` is an object that executes as an instance of the `Thread` class and, therefore, to make a thread functional, you must instantiate it, irrespective of the way it is created. Knowing the different constructors of the `Thread` class is important. The constructors of the `Thread` class that are used to instantiate `threads` are:

- Thread() (Default Constructor)
- Thread(Runnable trgt)
- Thread(Runnable trgt, String threadname)
- Thread(String threadname)

The constructor to instantiate a thread varies depending on the way it is created. When a thread is created by extending the `Thread` class, you can instantiate by using the following code snippet:

```
SampleThread thrd=new SampleThread();
```

In the preceding code snippet, the `SampleThread` class is the class that extends the `Thread` class and `thrd` is an instance of this class. Similar to other objects in `Java`, the `thrd` object is instantiated by using the `new` keyword.

However, if a `thread` class is created by implementing the `Runnable` interface, the instantiation of the `thread` is performed in the following steps:

1. First, create an object of the class that is implementing the `Runnable` interface
2. Then, get an instance of the `Thread` class and pass the object created in the first step while instantiating the object of the java.lang.Thread class

The following code snippet shows the implementation of the preceding steps:

```
class SampleThread implements Runnable
{
  public void run() {
      System.out.println("The code to perform specific function by thread");
  }
}
public class MyClass
{
  public static void main(String ar[]) {
      SampleThread thrd=new SampleThread();
      Thread T=new Thread(thrd);
  }
.
.
}
```

In the preceding code snippet, first, the thrd object of the `SampleThread` class is created, and then after creating an instance, `T`, of the `Thread` class, the `thrd` object is passed to the Thread class as a parameter.

In the code snippet, notice that when the thread object is created, the instance of the class that implements the `Runnable` interface is passed as an argument. This is because the `run()` method is implicitly invoked only when you create a class by extending the `Thread` class. However, in the preceding code snippet, you create a thread by implementing the `Runnable` interface. Therefore, you need to invoke the `run()` method explicitly. The explicit invocation of the `run()` method is achieved by passing the object of the class that implements the `Runnable` interface as an argument when the object of the `Thread` class is created. The object passed as an argument is sometimes called target or target `Runnable`.

You have just learned how to define and instantiate a thread. However, the thread is not functional and will not serve any purpose unless it invokes the start() method. Let's therefore learn how to start a thread.

6.4 Starting a Thread

When a thread is instantiated, it is in the new state (the states of a thread are discussed later in this chapter). After instantiation, you need to invoke the start() method to make it functional. The following code snippet shows how the start() method is invoked by a thread object:

```
thrd.start(); //Thread named thrd is started
```

Here, the start() method is invoked by the thrd object. The following events place when the start() method of a thread object is invoked:

- A new thread of execution starts with a new call stack
- The state of the thread changes from the new state to the runnable state
- The run() method is executed when the thread gets its turn

Remember that the start() method is always invoked by the instance of the Thread class and not by the objects of the class implementing the Runnable interface.

Listing 1 shows a complete example of defining, instantiating, and starting a thread (you can find the MyThread.java file on the CD in the code\chapter6 folder):

▶ **Listing 1: Defining, Instantiating, and Starting a Thread**

```
class SampleThread implements Runnable
{
 public void run()
 {
     for(int i=0;i<5;i++)
     {
         System.out.println("Hello World");
     }
 }
}
public class MyThread
{
 public static void main(String ar[])
 {
     SampleThread thrd=new SampleThread();
     Thread newThread=new Thread(thrd);
     newThread.start();
 }
}
```

In Listing 1, the SampleThread class is created by implementing the Runnable interface. In the main() method, an instance of this class is created and passed as an argument when the instance of the Thread class is created. The start() method is then invoked by the newly created instance, newThread, of the Thread class. The output of Listing 1 displays Hello World five times, as shown in Figure 1:

▲ **Figure 1: Output of the MyThread Program**

As shown in Figure 1, when the `start()` method is invoked on a `Thread` object, it changes its state from `new` to `runnable` and calls the `run()` method. Consequently, all the statements inside the `run()` method are executed.

TEST YOUR KNOWLEDGE

Q1. Write a program to create and use a thread.

Ans.

```
class SampleThread implements Runnable
{
 public void run()
 {
     for(int i=0;i<5;i++)
     {
         System.out.println("Thread Test");
     }
 }
}
public class TestYourKnowledge1
{
 public static void main(String ar[])
 {
     SampleThread thrd=new SampleThread();
     Thread newThread=new Thread(thrd);
     newThread.start();
 }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge1.java
D:\code\chapter6>java TestYourKnowledge1
```

Output:

```
Thread Test
Thread Test
Thread Test
Thread Test
Thread Test
```

Starting and Running Multiple Threads

In the preceding section, a single child thread with the main thread is created. You can also create multiple threads in a program. When you create multiple threads, it is important to understand the behavior of the program. A program with multiple threads show the following characteristics:

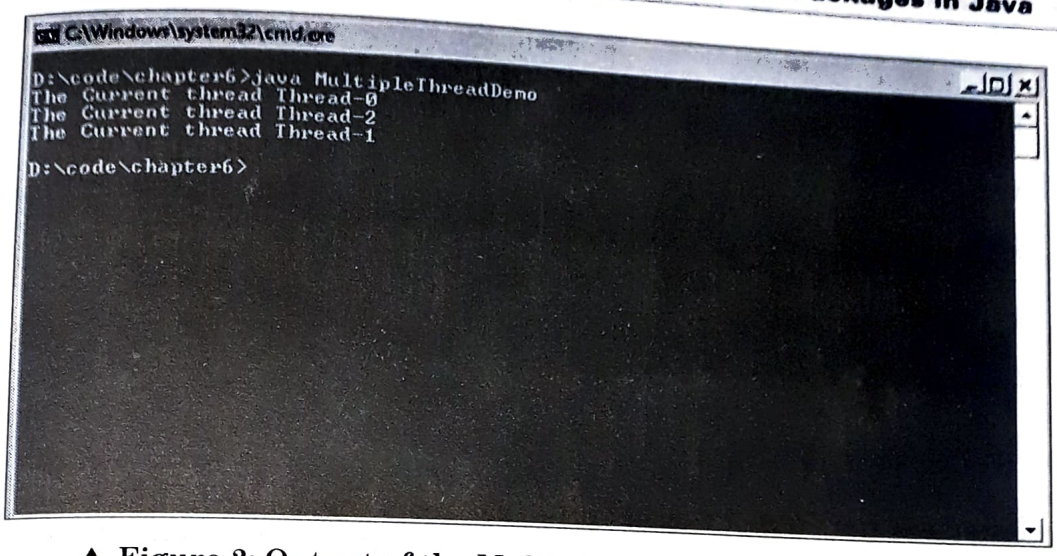
- Threads do not necessarily execute in the sequence in which they invoke the start() method
- If a thread is started, it is not guaranteed that it will keep executing until it finishes its task
- A single runnable instance can be used to create multiple threads

When a program with multiple threads is executed multiple times, it is not necessary that all the results match. This is because the order of execution of threads is not fixed. Listing 2 shows an example of how multiple threads are created and executed in a program (you can find the `MultipleThreadDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 2: Creating Multiple Threads**

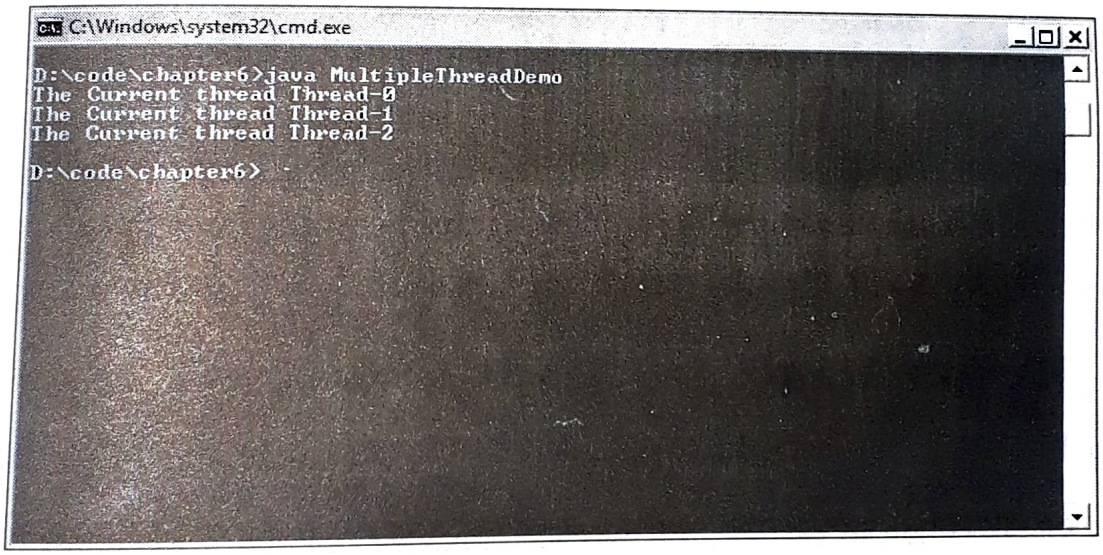
```
class Test implements Runnable

{
 public void run()

 {
     System.out.println("The Current
     thread"+Thread.currentThread().getName());

 }

}
public class MultipleThreadDemo
{
 public static void main(String ar[])

 {
     Test t=new Test();
     Thread t1=new Thread(t);
     Thread t2=new Thread(t);
     Thread t3=new Thread(t);
     t1.start();
     t2.start();
     t3.start();
}

}
```

In Listing 2, three `threads: t1, t2,` and `t3` are created. The default names of the `threads` are retrieved within the `run()` method and displayed by the print statement. After compiling and executing Listing 2, the output displayed is shown in Figure 2:



▲ **Figure 2: Output of the MultipleThreadDemo.java Program**

Figure 2 shows the result of executing multiple threads. It is not necessary that you get the same output because, when multiple threads are executed, their order of execution is not guaranteed. When you execute the MultipleThreadDemo program multiple times, you may get a different output each time. Figure 3 shows a different output of the MultipleThreadDemo program:



▲ **Figure 3: Different Output of the MultipleThreadDemo Program**

Compared to the output shown in Figure 2, the output shown in Figure 3 is different, although the same program is executed.

All the threads created in a program have their associated states and they change their states according to the methods invoked by them.

TEST YOUR KNOWLEDGE

Q2. Write a program to create and use multiple threads.

Ans.

```
class Test implements Runnable
{
 public void run()
 {
     System.out.println("The Current thread
```

```
            "+Thread.currentThread().getName());
    }
}

public class TestYourKnowledge2
{
    public static void main(String ar[])
    {
        Test t=new Test();
        Thread t1=new Thread(t);
        Thread t2=new Thread(t);
        Thread t3=new Thread(t);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge2.java
D:\code\chapter6>java TestYourKnowledge2
```

Output:

```
The Current thread Thread-0
The Current thread Thread-2
The Current thread Thread-1
```

Let's discuss thread states and how a thread changes from state to another in the following section.

5 Thread States and Transitions

Threads are scheduled for execution by the thread scheduler, which is a part of the JVM. Threads are stored in a runnable pool and the scheduler selects the threads that are eligible for execution in random order. The runnable pool is a collection of threads that are in the runnable state. The thread selected by the scheduler starts executing and it dies soon as the execution of the run() method is complete. Apart from new, running, and dead states, threads also have some intermediate states.

Thread States

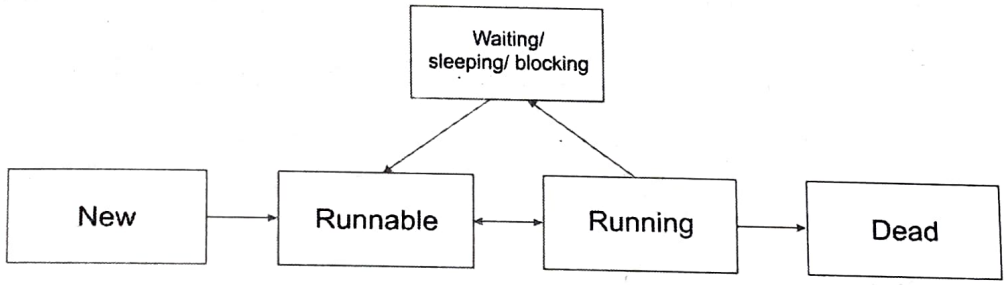
When an instance of the Thread class is created, you need to invoke different methods to make the thread functional. In other words, when you create a thread, it is only an instance of the Thread class instead of an independently executed thread. To make the thread eligible for execution, the start() method of the Thread class is invoked by a Thread object. After invoking the start() method, the thread moves to its runnable state.

To work with threads in a program, you need to identify the thread state. Let's discuss how to identify the thread state in the following section.

Identifying Thread States

A thread in a program is a path of execution, which can be in any one of the following five states:

- **New:** A thread is known to be in the new state when it has been instantiated but the start() method has not been invoked yet on the instance. In other words, the Thread object exists but it is not a thread of execution (that is, it cannot execute any instruction) as it is not yet alive.
- **Runnable:** Threads on which the start() method has been invoked and which are eligible to run but the thread scheduler has not yet selected them for execution are said to be in the runnable state. Threads in the runnable state are kept inside the runnable pool from where the scheduler selects them for execution. Runnable threads are also called live threads. A thread attains the runnable state the first time by the invocation of the start() method. A thread can also come back to the runnable state from a blocked, waiting, or sleeping state.
- **Running:** When a thread is selected by the thread scheduler from the runnable pool for execution, it achieves the running state. This is the state the thread performs its actual functions. A thread can come into the runnable state from the running, waiting, or new states. However, a thread can come into the running state only from the runnable state. Figure 4 shows the ways in which a thread can move from one state to another:



▲ **Figure 4: States of Threads**

- **Waiting/blocking/sleeping:** Here, three different states are combined together as they show a common behavior, that is, though the threads in these states are alive, they are not eligible to run. In other words, the threads are not runnable and are not selected by the scheduler for execution. The threads in the waiting/blocking/sleeping state may return to the runnable state depending on the occurrence of events in the program. A thread is said to be in the waiting state when its run() method causes it to wait while other threads are performing their functions. After a thread completes its task, it notifies the waiting thread, which then moves to the running state. A thread appears in the blocking state due to the unavailability of shared resources, such as I/O resources or Object locks. It returns to the running state after the required resource or Object lock becomes available. The sleeping state of a thread is caused by the code inside its run() method, which instructs it to sleep for some time due to various reasons, such as the execution of another thread and non-availability of resources. The sleeping thread becomes runnable after the sleeping time expires.
- **Dead:** A thread is said to be in the dead state when it has completed executing its run() method. In the dead state, a thread is just a Thread object and not a separate thread of execution. A dead thread is not alive and therefore cannot be brought back into any other state. An attempt to call the start() method on a dead thread causes a runtime exception.

All the threads created in a program have their own priority of execution. What these priorities are is discussed in the next section.

Thread Priorities

In a Java program, priorities are assigned to threads, based on which the thread to be executed first is decided. In case of multiple threads, thread priorities improve the efficiency of the program by determining which threads need to be executed. Thread priorities are usually represented by numbers between 1 and 10. The thread that has the highest priority will be the current thread of execution. The thread scheduler selects the thread that has the highest priority from the runnable pool, and then executes the thread. When a higher priority thread joins the runnable pool, then the currently executing thread is sent back to the runnable pool and the thread of higher priority is executed. Therefore, the currently executing thread is always the thread that has the highest priority among the threads available in the runnable pool. In case threads that have the same priority join the runnable pool, then their execution will depend on the following two aspects:

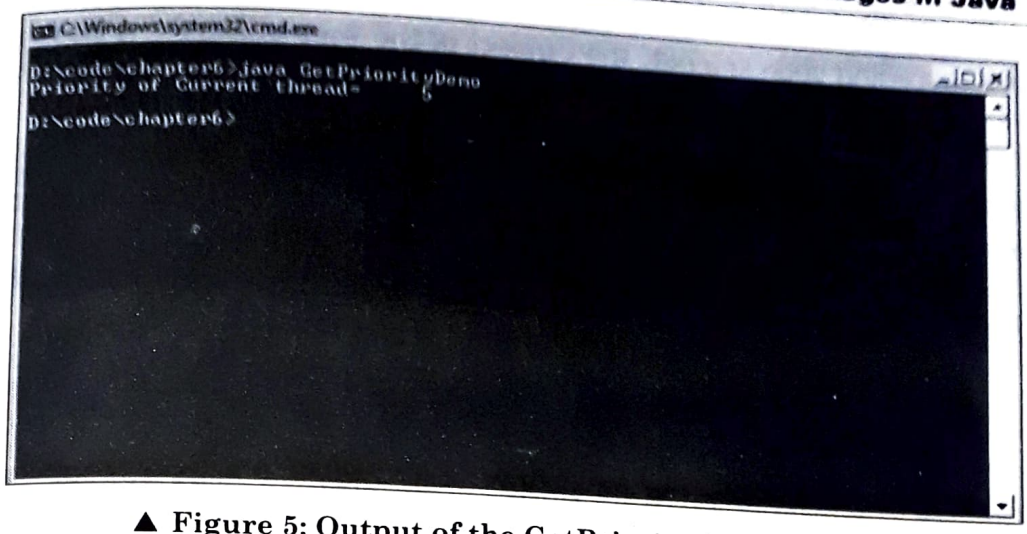
- The scheduler picks a thread randomly to run and allows it to complete its execution
- Execution time is divided and each thread is given equal time to execute in a round robin fashion.

In case we do not assign any priority to a thread, a default priority is assigned to it. This priority can be retrieved by invoking the static getPriority() method of the Thread class. Listing 3 shows how to invoke the `getPriority()` method (you can find the `GetPriorityDemo.java` file on the CD in the code\chapter6 folder):

▶ Listing 3: Getting the Default Priority of a Thread

```
class Test implements Runnable
{
 public void run() {
     System.out.println("Priority of Current thread=  "+Thread.currentThread
 ().getPriority());

 }
}
public class GetPriorityDemo
 {
 public static void main(String ar[])
 {
     Test t=new Test();
     Thread t1=new Thread(t);
     t1.start();
 }
}
```

In Listing 3, the t1 thread is created and the t object of the runnable class Test is passed as an argument. Within the `run()` method, the priority of the thread is retrieved by invoking the `getPriority()` method. The print statement inside the `run()` method displays the priority of the t1 thread, as shown in Figure 5:



▲ **Figure 5: Output of the GetPriorityDemo Program**

The output of Listing 3 displays the default priority 5, as shown in Figure 5. All the threads created in a program have the default priority 5.

TEST YOUR KNOWLEDGE

Q3. Write a program to get the priority of a thread.

Ans.

```
public class TestYourKnowledge3 implements Runnable
{
 static Thread t1;
 public void run() {
     System.out.println(Thread.currentThread().getName());
 }
 public static void main(String args[]) {
     TestYourKnowledge3 t=new TestYourKnowledge3 ();
     t1=new Thread(t);
     t1.start();
     System.out.println(t1.getPriority());
 }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge3.java
D:\code\chapter6>java TestYourKnowledge3
```

Output:

```
5
Thread-0
```

Apart from a default priority, you can also assign the priority of a thread in a program by invoking the `setPriority()` method on the Thread object, as shown in the following code snippet:

```
Test t=new Test();
Thread t1=new Thread(t);
t1.setPriority(4);
t1.start();
```

In the preceding code snippet, the t object of the Test runnable class is created and passed as an argument when the Thread object t1 is created. The priority of the t1 thread is set to 4 by invoking the setPriority() method.

The range of priority that can be assigned to a thread varies from one JVM to other. Few JVMs may support only the range of numbers between 1 and 5. In such a case, if you have created 10 threads, then a few of them will have the same priority. The Thread class has the following static final variables (constants) that define the range of thread priorities:

- MIN_PRIORITY: Invoked as Thread.MIN_PRIORITY(1)
- NORM_PRIORITY: Invoked as Thread.MIN_PRIORITY(5)
- MAX_PRIORITY: Invoked as Thread.MIN_PRIORITY(10)

The thread scheduler plays an important role in the life cycle a thread. Based on the priority of threads, it selects the threads for execution from the runnable pool of threads. The thread with the highest priority is executed first. Let's discuss thread scheduler in detail.

Thread Scheduler

The component of the JVM that decides the order in which threads are executed is called the thread scheduler. A system with a single processor, when executing a program, executes only a single thread at a particular point even though the program may have multiple threads. This means that even if there are multiple active threads, the resources are allocated only to the thread that is executing. The thread scheduler selects a thread for execution from the runnable pool and sends back the executing thread to the pool depending on the priority of the thread and the code inside the run() method of the thread.

In a Java program, you need not write explicit code to control the thread scheduler as it is the part of JVM. In most cases, the thread scheduler maps a thread of your program to the thread of the operating system of your machine. You cannot control the sequence of thread execution; rather, you can invoke methods of the Thread class to influence the thread scheduler. Now, let's discuss the methods that can affect the activities of the thread scheduler.

Running and Yielding

Sometimes, the running pool receives a thread of higher priority than the thread that is currently running. In such situations, the thread scheduler sends the currently running thread back to the runnable state and the thread that has the higher priority starts executing. To send a running thread back to the runnable state, the static method yield() of the Thread class is invoked on the current Thread object, as shown in the following code snippet:

```
Thread.yield()
```

In the preceding code snippet, the yield() method is invoked on the Thread object. Invoking the yield() method on a Thread object pauses the execution of the current thread and allows another thread of equal or higher priority to start its execution. Invoking the yield() method on a Thread object does not guarantee the non-execution of the thread; rather, the thread scheduler can select the thread on which the yield() method has been invoked for further execution. Moreover, invocation of the yield() method on a Thread

object does not move it to the `waiting`, `sleeping`, or `blocking` state; rather, it sends the `Thread object` to the `runnable` state, which can be resumed to the `running` state later.

In some situations, you need to deliberately make a thread sleep to make it release certain resources for a particular interval of time. Let's now discuss how a thread is deliberately made to sleep.

Sleeping and Waking Up

Sometimes you need to make a thread sleep for a particular interval of time. For example, consider the scenario of the train reservation system, wherein the reservations status must be updated frequently to know the current status. However, the status need not be updated continuously as it is not likely to change at every instant. The status changes only when a reservation or cancellation occurs. Therefore, it is better to get the reservation status after a certain interval of time. In such a situation, you can use a thread that keeps track of the reservation status after a particular interval of time. To execute a thread after a certain interval of time, make it `sleep` by invoking the `static` method, `sleep()`, of the `Thread` class. The `sleep()` method is invoked by a `Thread` object, as shown in the following code snippet:

```
try{
 Thread.sleep(500);
}catch(InterruptedException ex){}
```

In the preceding code snippet, the `sleep()` method is invoked on the `Thread` object. It causes the currently executing thread to sleep for 500 milliseconds. After the specified time interval expires, the thread wakes up and resumes its execution.

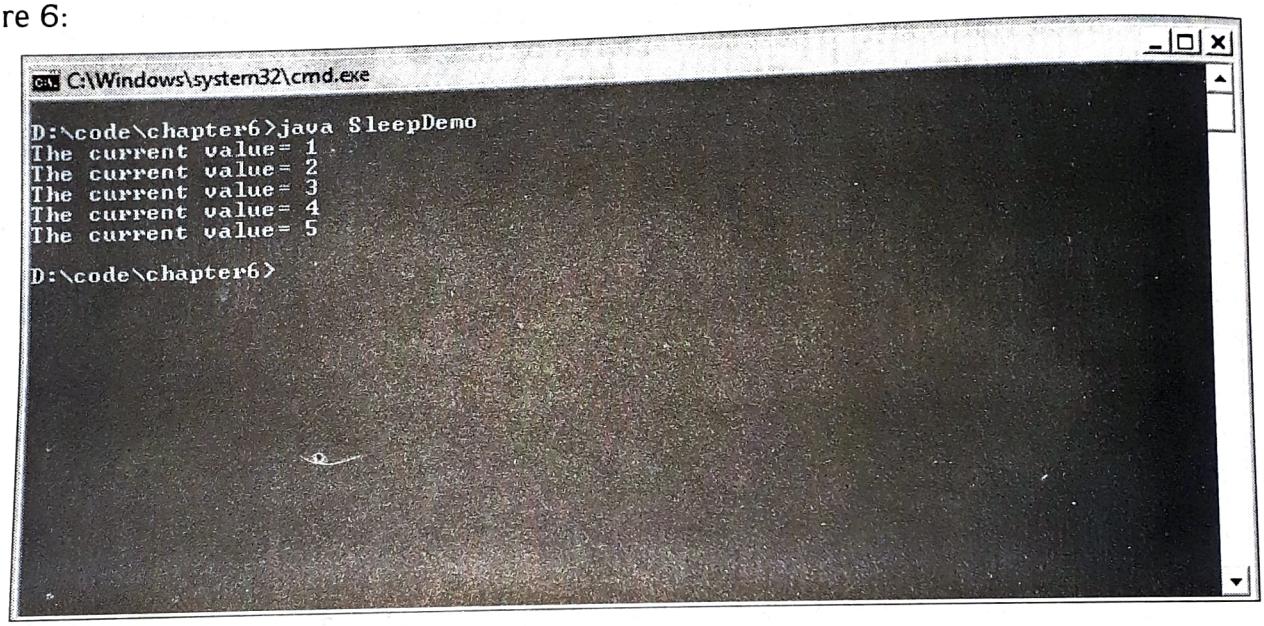
You need to make a thread sleep when multiple threads are running so that shared resources can be acquired by other threads. The `sleep()` method can throw an `InterruptedException` (checked exception) when other executing threads interrupt the sleeping thread. Therefore, as shown in the preceding code snippet, the `sleep()` method is always written inside the `try/catch` block anywhere in the program. Listing 4 shows an example of a `Thread` object invoking the `sleep()` method (you can find the `SleepDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 4: Threads Invoking the sleep() Method**

```
class Test implements Runnable
{
 public void run()
 {
     for(int i=1; i<=5;i++)
     {
         System.out.println("The current value= "+i);
         try
         {
             Thread.sleep(1000);
         }catch(InterruptedException E)
         {
             System.out.println("Thread Interrupted");
         }
     }
 }
}
```

```
public class SleepDemo
{
  public static void main(String ar[])
  {
      Test t=new Test();
      Thread t1=new Thread(t);
      t1.start();
  }
}
```

In Listing 4, the `t1` thread is instantiated and started, and the `sleep()` method is invoked inside the `run()` method, which causes the currently executing `t1` thread to pause for 1 second (1000 milliseconds). When you compile and execute the code shown in Listing 4, you notice a pause of 1 second while displaying the output from 1 to 5, as shown in Figure 6:



▲ **Figure 6: Output of the SleepDemo Program**

Figure 6 shows the output of Listing 4 and displays numbers from 1 to 5, with a pause of 1 second after each number.

When multiple threads are instantiated and used in a program, the output is not predictable with the invocation of the sleep() method. The reason is that when a thread comes across the `sleep()` method, it goes to sleep for the specified interval of time (milliseconds) until it is interrupted by the other thread (in the case of an interruption, it throws the `InterruptedException` exception). After the expiry of the specified time-interval, the thread resumes its `runnable` state, not the `running` state. In other words, when the time specified in the `sleep()` method is completed, the thread is eligible for execution. However, the execution of the thread can be delayed when a thread of higher priority needs to be executed. Therefore, it is not guaranteed that a thread will start executing just after the completion of the time-interval specified in the `sleep()` method. You must know that the static method `sleep()` always causes only the current thread to go to sleep; it does not affect the states of other threads.

TEST YOUR KNOWLEDGE

Q4. Write a program to demonstrate the use of the sleep() method.

Ans.

```
class Test implements Runnable
{
  public void run()
  {
      for(int i=1; i<=5;i++)
      {
          System.out.println("The current value= "+i);
          try {
              Thread.sleep(1000);
          }catch(InterruptedException E)
          {
          System.out.println("Thread Interrupted");
          }
      }
  }
}
public class TestYourKnowledge4 {
  public static void main(String ar[])
  {
      Test t=new Test();
      Thread t1=new Thread(t);
      t1.start();
  }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge4.java
D:\code\chapter6>java TestYourKnowledge4
```

Output:

```
The current value= 1
The current value= 2
The current value= 3
The current value= 4
The current value= 5
```

Now, let's discuss the join() method used to influence the thread scheduler to change the state of the current thread.

▪ Joining

In a `Java` program, the main `thread` is generally the last thread to be terminated. However, this is not always true. You can force the main `thread` to wait until the child thread finishes its execution by invoking the `sleep()` method inside the `main()` method. Now, if the time period of the `sleep()` method for the child thread is longer than that of the main `thread`, then the main `thread` terminates before the child thread. The thread class provides the `join()` method, which can be used to make the current thread wait until the thread that has

invoked the `join()` method completes its execution. The `join()` method is non-static and always invoked by an instance of the Thread class.

Listing 5 shows an example of executing threads without the use of the `join()` method (you can find the `WithoutJoinDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 5: Executing Threads Without Using the join() Method**

```
class Test implements Runnable
{
  public void run() {
      for(int i=1; i<=5;i++) {
          System.out.println("The current value= "+i);
      }
      System.out.println("Child Terminated");
  }
}
public class WithoutJoinDemo {
  public static void main(String ar[]) {
      Test t=new Test();
      Thread t1=new Thread(t);
      t1.start();
      System.out.println("Main Terminated");
  }
}
```

In Listing 5, the t1 thread is created by using the runnable object `t` of the runnable class Test, and then the start() method is invoked by the `t1` object which, in turn, invokes the run() method. Note that in this case, the main thread terminates before the termination of the child thread t1, as shown in Figure 7:

▲ **Figure 7: Output of the WithoutJoinDemo Program**

Figure 7 shows that the main `thread` is terminated before the termination of the child thread. To make the main `thread` wait for the termination of the child thread, you need to invoke the `join()` method on the child thread. The `join()` method is invoked by using the following code snippet:

```
Test t=new Test();
Thread t1=new Thread(t);
try
```

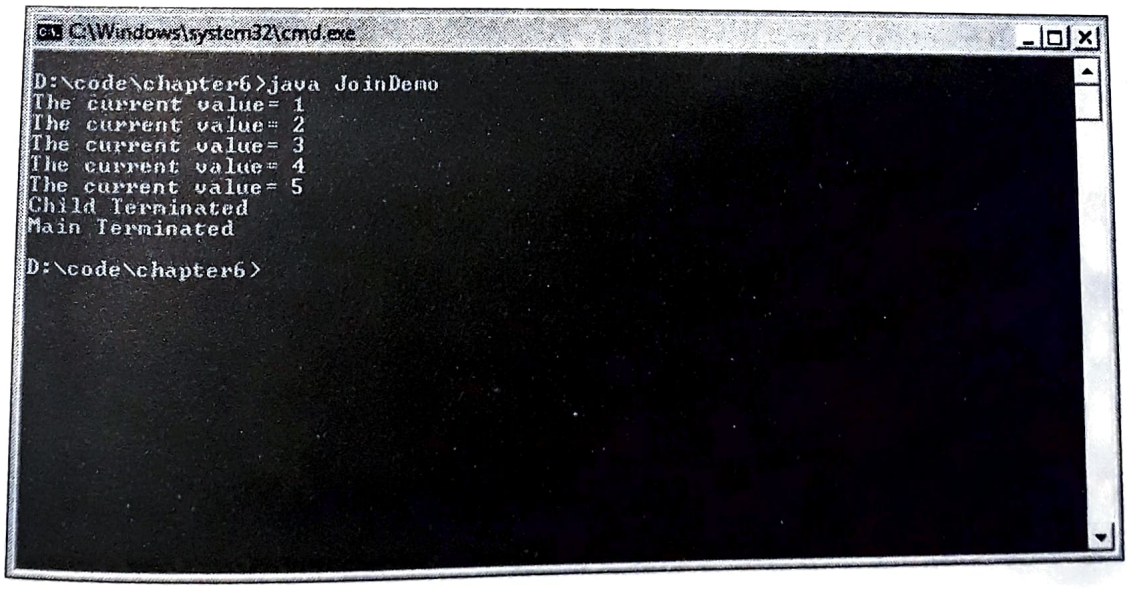
```
    {
      t1.join();
    }catch(InterruptedException e){}
```

In the preceding code snippet, the `join()` method is invoked on the `t1` Thread object, which makes the current thread wait until the `t1` thread finishes its execution. The `join()` method generates an `InterruptedException` exception and therefore must be invoked in the `try/catch` block. Listing 6 shows the invocation of the `join()` method (you can also find the `JoinDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 6: Invoking the `join()` Method to Make the Main Thread Wait for the Child Thread**

```
class Test implements Runnable {
  public void run() {
      for(int i=1; i<=5;i++)
      {
          System.out.println("The current value= "+i);
      }
      System.out.println("Child Terminated");
  }
}
public class JoinDemo {
  public static void main(String ar[]) {
      Test t=new Test();
      Thread t1=new Thread(t);
      t1.start();
      try
      {
          t1.join();
      }catch(InterruptedException e){}
      System.out.println("Main Terminated");
  }
}
```

In Listing 6, the `join()` method is invoked by the `t1` thread object, which causes the main `thread` to wait until the `t1` child thread finishes its execution. The child thread `t1` terminates before the main thread, as shown in Figure 8:



▲ **Figure 8: Output of the JoinDemo Program**

Figure 8 shows that the main thread does not terminate until the t1 child thread finishes its execution.

The Thread class also provides the following two overloaded versions of the join() method:

- **void join(long milliseconds)**: The current thread waits for the calling thread for the specified time-interval
- **void join(long millisecond, int nanosecond)**: The current thread waits for the calling thread for the specified milliseconds and nanoseconds

If the current thread takes long to terminate, then the other thread needs to wait unnecessarily. This waiting can be avoided by invoking the overloaded version of the join() method. The overloaded version of the join() method makes the current thread wait for a specified period of time. If the calling thread does not complete its process in the specified period of time, the current thread stops waiting and comes in the `runnable` state.

TEST YOUR KNOWLEDGE

Q5. Write a program to demonstrate the use of the join() method.

Ans.

```
class Test implements Runnable
{
 public void run()
 {
     for(int i=1; i<=5;i++)
     {
         System.out.println("The current value= "+i);
     }

     System.out.println("Child Terminated");
 }
}

public class TestYourKnowledge5
{

 public static void main(String ar[])
 {
     Test t=new Test();
     Thread t1=new Thread(t);
     t1.start();
     try
     {
        t1.join();
     }catch(InterruptedException e){}

     System.out.println("Main Terminated");
 }

}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge5.java
D:\code\chapter6>java TestYourKnowledge5
```

Output:

```
The current value= 1
The current value= 2
The current value= 3
The current value= 4
The current value= 5

Child Terminated
Main Terminated
```

If a thread in a program is responsible for some input/output (I/O) operation, then it can be put in the `blocking` state until the I/O resources are available. Let's discuss how to do this in the next section.

Blocking for I/O

Threads in a program can also be used to perform I/O operations. When multiple threads share resources, it is not guaranteed that a thread eligible for execution gets the needed resources. This is because other threads may be using the shared resources at the time and the thread eligible for execution and trying to access the shared resources is blocked temporarily. The blocked thread attains the `runnable` state after the shared resources are free.

The threads that are instantiated and used in a program have a definite life cycle and finally get terminated at the end of their life cycle.

Thread Termination

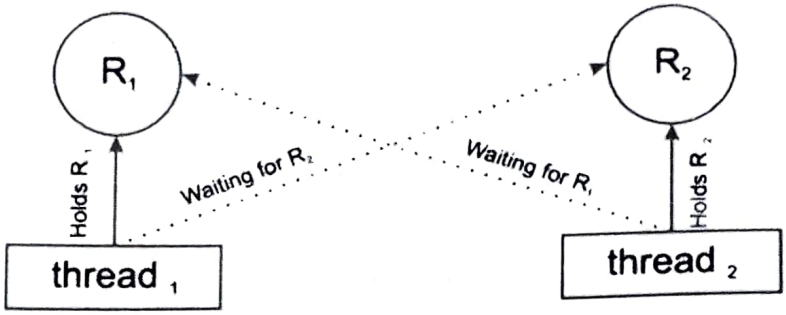
A thread is moved to the dead state from either the runnable or running state. Moreover, a thread is said to be in the dead state after it completes its run() method, which also implies that the thread is terminated. If any other thread or event does not interrupt a thread, then it terminates normally. However, if a running thread is interrupted, it throws the InterruptedException exception and is terminated. After a thread is moved into the dead state, it cannot be made alive even after the invocation of the start() method.

When an attempt to invoke the `start()` method is made on a dead thread, the `start()` method throws the `IllegalThreadStateException` exception. In Java, the invocation of methods that are not allowed in a particular thread state throw the `IllegalThreadStateException` exception.

With this, we come to another important concept related to threads, that is, deadlocks.

Deadlocks

When multiple threads are used in a program, they share the resources. A particular resource is allocated to a thread that acquires the Object lock for that resource. `Object locks` are variables that are acquired by any thread from the runnable pool. Deadlock is a situation where two threads wait for the Object locks acquired by each other. The Object locks are not released by the respective threads before completing their execution. This situation, where threads are waiting for the Object lock forever, leads to a deadlock. Figure 9 shows a `deadlock`:



▲ **Figure 9: Threads Showing Deadlock**

Figure 9 shows that the `thread1` and `thread2` threads are in a deadlock. The `thread1` thread holds the lock for the `R1` resource and waits for the `R2` resource, which is acquired by the `thread2` thread. At the same time, the `thread2` thread waits for the lock of the `R2` resource, which is held by the `thread1` thread. Therefore, both the threads keep waiting, leading to a deadlock.

6.6 Code Synchronization

When multiple threads are used in a Java program, it is possible that more than one thread accesses the same instance of a class. When multiple threads try to modify the state of an object simultaneously without intimating the other threads, it leads to an inconsistency in the state of the object.

For example, consider a scenario of the railway reservation system, where two passengers are trying to reserve seats from Delhi to Mumbai in Rajdhani Express. Although the passengers are at different locations, they have access to the same railway reservation system. Now, let's suppose that both passengers start their reservation process at 1 pm and observe that two seats are available. The first passenger reserves these two seats and simultaneously the second passenger reserves one seat. Now, the available numbers of seats were two and the reserved seats are three. This leads to inconsistency due to the asynchronous access to the railway reservation system. In this scenario, both the passengers can be considered as two threads, and the reservation system can be considered as a single object, which is modified by these two threads asynchronously.

This problem of inconsistency is known as the race condition, in which multiple threads access the same object and modify the object's state inconsistently. The solution to this problem is to provide a mechanism in which, if one thread is modifying an object's state, the other thread does not have access of the same object at that point of time. In other words, the modification of the state of an object must be `synchronized` and `atomic`.

Locks

Synchronization of code in a `Java` program is done with the help of `locks`. Every object in the Java programming language has a default `object lock`, which can be used for synchronization. Threads gain access to a shared object of a program by first acquiring the `lock` associated with it. After a thread acquires the lock associated with an object, other threads are prevented from acquiring the lock associated with the object at that point of time. This guarantees that only one thread modifies the state of the object at a time. The Object lock, also known as monitor, allows only one thread to use the shared resources at a time. In this way, the lock implements mutual exclusion.

In a Java program, the access to the shared resources (objects) is synchronized by the object lock mechanism. The rules that need to be followed when synchronizing shared resources are:

- A thread must acquire the object lock associated with a shared object before it uses the shared object. The runtime system ensures that if a thread has the lock of the shared object, it will not allow another thread to access the shared object. If a thread needs to access the shared resource at the same time, it is blocked and has to wait until the resource becomes available.
- When a thread completes its execution and finishes modifying the shared resource, the runtime system again ensures that the lock of the shared resource has been relinquished by the thread using the resource. The resource is then assigned to the thread waiting for it.
- Methods or blocks can be synchronized, but not variables or classes.
- Only one lock is associated with each object/shared resource.

In Java, code synchronization can be achieved in the following two ways:

- Synchronized Methods
- Synchronized Blocks

Now, let's discuss these in detail in the following sections.

Synchronized Methods

In a `Java` program, methods associated with an object can be invoked by multiple threads. The methods that modify the state of the object must be synchronized. A method can be `synchronized` by using the `synchronized` keyword before the name of the method while defining the method. After a method is `synchronized`, a thread needs to acquire the `object lock` to invoke the method. If the `object lock` is not available, the calling thread is blocked and it has to wait until the lock becomes available. After a thread completes executing the synchronized method, it relinquishes the object `lock` and allows the other threads waiting for this lock to proceed.

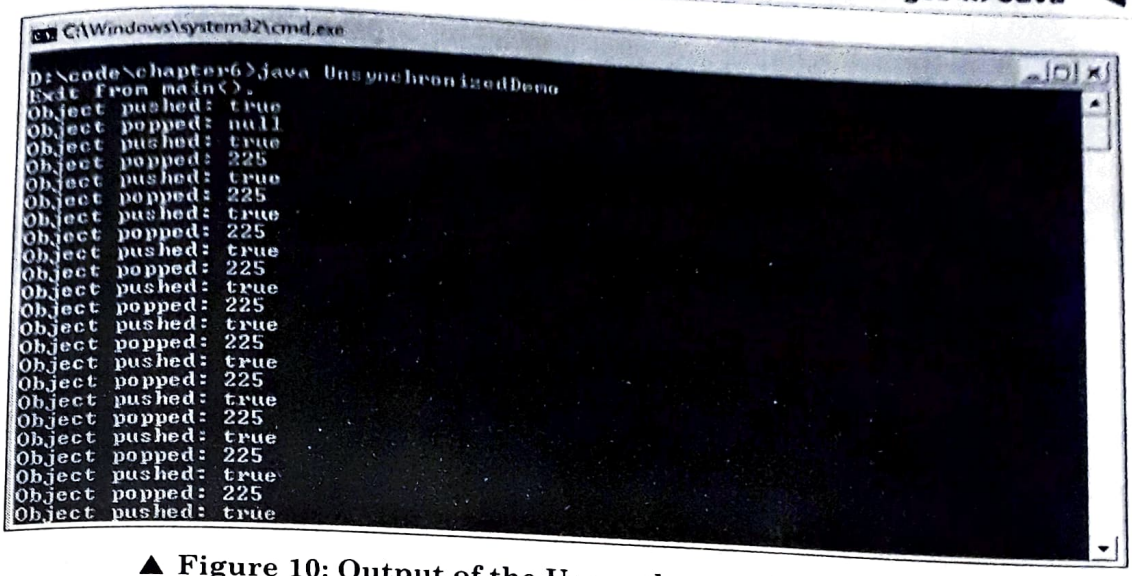
Synchronizing methods is important in situations where the methods invoked by threads modify the state of the object. The concurrent execution of such methods leaves the object in an inconsistent or corrupt state. Let's consider an example of `stack` implementation, in which two threads are used to insert and retrieve data to and from the stack. However, if the insert and retrieve operations are performed on the same stack, then problems may occur due to the inconsistent state of the stack. Listing 7 shows an example of the implementation of the stack causing a runtime exception because of unsynchronized method invocation (you can find the `UnsynchronizedDemo.java` file on the CD in the code\chapter6 folder):

▶ Listing 7: Two Threads Invoking Unsynchronized Methods

```
class DemoStack {
  private Object[] stck;
  private int top;
  public DemoStack(int capacity) {
      stck= new Object[capacity];
      top= -1;
  }
  public boolean push(Object obj)   //unsynchronized push method
  {
```

```
            if (top>=stck.length-1)
            return false;
            ++top;
            try {
                Thread.sleep(500);
            } catch (Exception E) { }
            stck[top] = obj;
            return true;
        }
        public Object pop()        //unsynchronized pop method
        {
            if (top<0)
            return null;
            Object obj = stck[top];
            stck[top] = null;
            try {
                Thread.sleep(500);
            } catch (Exception E) { }
            top--;
            return obj;
        }
    }
    public class UnsynchronizedDemo
    {
        public static void main(String ar[])
        {
            final DemoStack stack = new DemoStack(10);
            (new Thread()      //First Thread calling push method
            {
                public void run() {
                    for(;;) {
                        System.out.println("Object pushed: " + stack.push(new
                        Integer(225)));
                    }
                }
            }).start();
            (new Thread()      //Second Thread calling pop method
            {
                public void run() {
                    for(;;) {
                        System.out.println("Object popped: " + stack.pop());
                    }
                }
            }).start();

            System.out.println("Exit from main().");
        }
    }
```

In Listing 7, two threads are instantiated, which invoke the `push()` and `pop()` methods of the DemoStack object stack. As these are non-synchronized methods, both the threads invoke both the methods asynchronously, causing a runtime exception. Figure 10 shows the output of Listing 7:



▲ **Figure 10: Output of the UnsynchronizedDemo Program**

The `ArrayIndexOutOfBound Exception` runtime exception is generated when Listing 7 is executed. This exception occurs due to the invocation of the `non-synchronized push()` and `pop()` methods, which modify the top of the stack. This runtime exception can be avoided by synchronizing the invocation of the `push()` and `pop()` methods, as shown in the following code snippet:

```
public synchronized boolean push(Object obj)    //synchronized push method
{
  if (top>=stck.length-1)
  return false;
  ++top;
  try {
      Thread.sleep(500);
  } catch (Exception E) { }
  stck[top] = obj;
  return true;
}
```

In the preceding code snippet, the `push()` method is synchronized by adding the synchronized keyword. Similarly, the `pop()` method can also be synchronized. Now, at a time, only one thread can invoke these methods after acquiring the `Object lock` of the stack object. Therefore, no inconsistency in the object state occurs and the code does not generate any runtime exception.

Along with `synchronized` methods, a class can also have `non-synchronized` methods, in which the data that you do not need to protect can be manipulated. The following facts need to be kept in mind while using `synchronized` and `non-synchronized` methods:

- There are greater chances of a deadlock when synchronized methods are used. Apart from synchronized methods, multiple threads can also invoke non-synchronized methods.
- If a thread has an object lock that goes to the sleep state, then it does not relinquish the object lock.
- If a thread is executing a synchronized method and needs to invoke another synchronized method, then it can get the object lock to invoke the method. After it finishes executing the methods, it relinquishes the locks.

- If a thread acquires an object lock for a particular object, any synchronized method associated with the object can be invoked by the thread without a new lock for the object.

In the Java programming language, static methods can also be synchronized. While synchronizing static methods, a thread needs to acquire the class lock instead of an object lock. This is because static methods belong to a class and not to the instances of a class. The static method can be synchronized as shown in the following code snippet:

```
public static synchronized void display() {
  System.out.println("This is synchronized method");
}
```

In the preceding code snippet, the static display() method is created as a synchronized method.

Synchronized Blocks

Synchronizing methods may result in a deadlock, which can be avoided by synchronizing certain block of codes instead of a complete method. When a block of code is synchronized, the thread executing the synchronized block needs to acquire the object lock of an arbitrary object. Generally, it is the current object whose lock is acquired by a thread to invoke the synchronized block, as shown in the following code snippet:

```
class SynchronizedBlockRep {
  void display() {
      System.out.println("This is not synchronized");
      synchronized(this)
      {
          System.out.println("This is synchronized block");
      }
  }
}
```

In the preceding code snippet, only the second print statement is synchronized, instead of the entire display() method.

Synchronizing blocks allows threads to have more than one object lock for code synchronization within a single object. If a thread acquires an object lock to execute a synchronized block, then other threads are blocked to wait for the object lock that is relinquished by the currently executing thread after it finishes its execution. The following facts need to be kept in mind while blocking threads:

- In a program, threads calling static synchronized methods in the same class can block each other
- Threads that call non-static synchronized methods in a class lock each other if they invoke the methods by using the same instance of the class
- A non-static synchronized method and a static synchronized method cannot block each other as a non-static method locks an instance of the class, whereas static methods lock the class itself

In case of a synchronized block, threads that synchronize the same object can block each other, whereas threads that synchronize different instances cannot block each other.

Next, we discuss the mechanism by which multiple threads interact with each other a program.

6.7 Thread Interaction

Multiple threads in a program are capable of interacting with each other and sharing certain information, such as their state and locking status. For example, a thread in the `waiting` state continuously checks whether an `object lock` for a synchronized method is available or not, which causes unnecessary wastage of time and resources. After the current thread releases the shared resources, it can notify the waiting threads about the status of the `lock`. Threads can interact with each other by using the following methods provided by the `Object class`:

- wait()
- notify()
- notifyAll()

These methods must be invoked from a synchronized context, that is, from a synchronized method or block. A thread can invoke the wait() and notify() methods on an object only when it has acquired the lock for the object. If the thread invoking the `wait()` method does not acquire the lock, it throws an `IllegalMonitorStateException` exception. As it throws a checked exception, you need to use this method from within the try/catch block. Consider the following code snippet, which shows the implementation of the `wait()` and `notify()` methods:

```
class Mythread extends Thread {
  public void run() {
      .
      .
      synchronized(this) {
          .
          .
          notify();
      }
  }
}
class ThreadDemo extends Thread {
  .
  .
  public void run() {
      ..
      synchronized(MyThread thrd) {
          ..
          try {
              thrd.wait();
          }catch(InterruptedException E){}
      }
  }
}
```

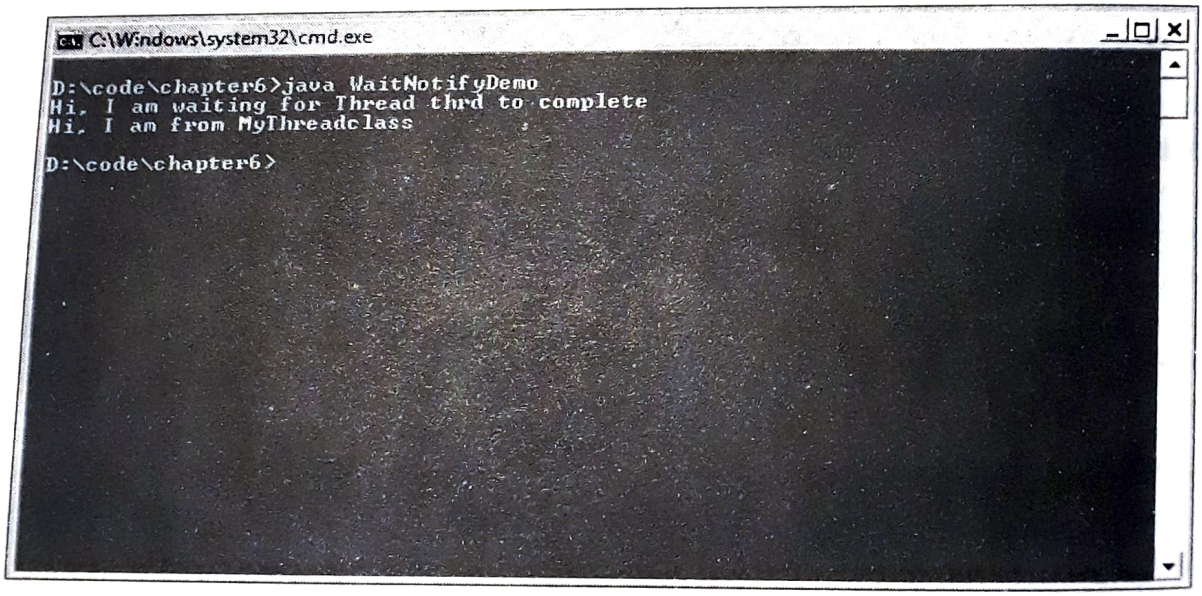
In the preceding code snippet, the MyThread and ThreadDemo classes have been declared. The `run()` method of the MyThread class invokes the `notify()` method, which acts as a response for the ThreadDemo class thread. Within the `run()` method of the `ThreadDemo` class, the `wait()` method is invoked on the object of the MyThread class, which makes the current thread wait until it is notified. In the preceding code snippet, the wait() and notify() methods are invoked from within the synchronized context. Moreover, the wait() method is

invoked from within the pair of the try/catch block, because the method generates the InterruptedException (checked exception) exception. The waiting thread may be interrupted by other threads or events and therefore generates this exception. Listing 8 shows a working example of the invocation of the wait() and notify() methods (you can find the WaitNotifyDemo.java file on the CD in the code\chapter6 folder):

▶ **Listing 8: Invoking the wait() and notify() Methods**

```
class MyThread extends Thread {
  String msg="Hi, I am from MyThread";
  public void run() {
      synchronized(this) {
          msg=msg+ "class";
          notify();
      }
  }
}
class WaitNotifyDemo extends Thread {
  public static void main(String ar[]) {
      MyThread thrd=new MyThread();
      thrd.start();
      synchronized(thrd)
      {
          try {
           System.out.println("Hi, I am waiting for Thread thrd to complete");
              thrd.wait();
          }catch(InterruptedException E){}
          System.out.println(thrd.msg);
      }
  }
}
```

When you compile and execute the code given in Listing 8, the output appears, as shown in Figure 11:



▲ **Figure 11: Output of the WaitNotifyDemo Program**

The wait() and notify() methods are non-static methods of the Object class. In other words, these methods are associated with instances of the Object class and not the

Object class itself. By executing the wait() method of the Object class, you can retrieve the list of waiting threads. The currently executing thread intimates the other threads about the availability of the lock by invoking the `notify()` method. For example, when a customer visits a bank, he gets a token with a token number, which means that the customer is added to the waiting list of the bank service object. The customer is notified of the service by displaying the token number at the service terminal. It is the same with objects and threads. objects represent bank services and threads represent the customers.

TEST YOUR KNOWLEDGE

Q6. Write a program to demonstrate use of the wait() and notify() methods.

Ans.

```
class MyThread extends Thread
{
 String msg="Hi, I am from MyThread";
 public void run()
 {
     synchronized(this)
     {
         msg=msg+ "class";
         notify();
     }
 }
}
class TestYourKnowledge6 extends Thread
{
 public static void main(String ar[]) {
     MyThread thrd=new MyThread();
     thrd.start();
     synchronized(thrd)
     {
         try {
System.out.println("Hi, I am waiting for Thread thrd to  complete");
             thrd.wait();
         }catch(InterruptedException E){}
         System.out.println(thrd.msg);
     }
 }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge6.java
D:\code\chapter6>java TestYourKnowledge6
```

Output:

```
Hi, I am waiting for Thread thrd to complete
Hi, I am from MyThreadclass
```

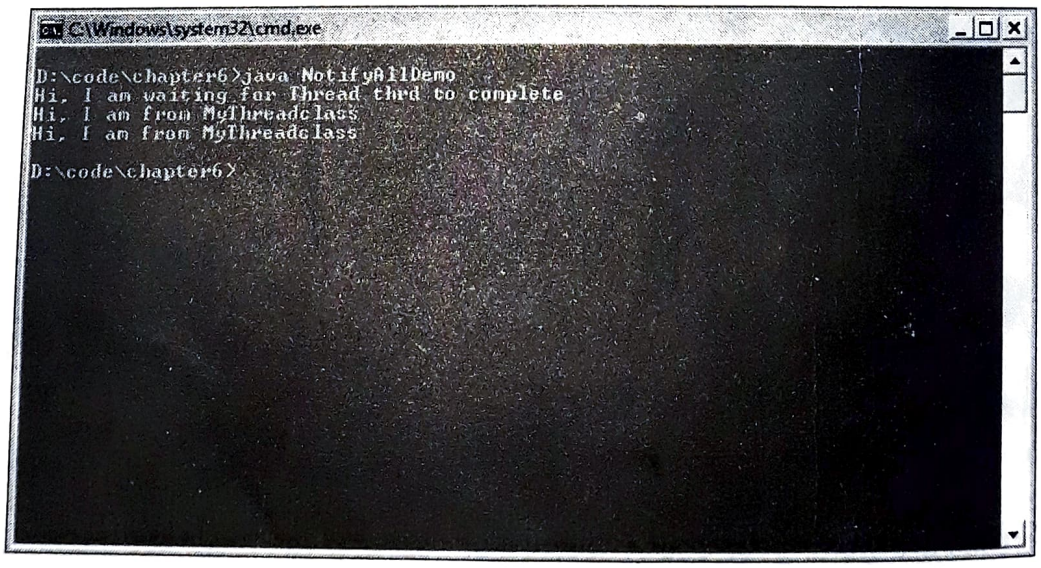
Apart from the `wait()` and `notify()` methods, the Object class also provides the `notifyAll()` method, which is similar to the `notify()` method. In a program, if multiple threads are waiting for a lock, instead of notifying the threads separately, you can invoke the `notifyAll()` method to notify all the threads simultaneously, after which no further

notification is required. All the notified threads compete for the lock and start executing after they get the lock. Listing 9 shows an example of invocation of the `notifyAll()` method (you can find the `NotifyAllDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 9: Invocation of the `notifyAll()` Method**

```
class MyThread extends Thread
 {
  String msg="Hi, I am from MyThread";
  public void run() {
      synchronized(this)
      {
          msg=msg+ "class";
          notifyAll();
      }
  }
}
class NotifyAllDemo extends Thread
{
  public static void main(String ar[])
  {
      MyThread thrd=new MyThread();
      MyThread thrd1=new MyThread();
      thrd.start();
      thrd1.start();
      synchronized(thrd)
      {
          try {
          System.out.println("Hi, I am waiting for Thread thrd to complete");
          thrd.wait();
          }catch(InterruptedException E){}
          System.out.println(thrd.msg);
          System.out.println(thrd1.msg);
      }
  }
}
```

The modified code of Listing 8 is shown in Listing 9. Instead of using the `notify()` method, the `notifyAll()` method is invoked. Figure 12 shows the output of Listing 9:



▲ **Figure 12: Output of the NotifyAllDemo Program**

Till now, you must have a good idea about threads and know how to create and use them in a program. Now, let's discuss another important concept, that is, packages in Java.

6.8 Working with Packages in Java

A package can be considered a directory or folder, which allows you to store various classes related to each other in an application. Packages resolve the problem of class name collision. In Java, you cannot create two classes with the same name. This means that you have to assign a different name for every new class. This, however, is a tedious task. To resolve this problem, you can use package.

To understand how packages resolve class name collision, let's consider an example. Suppose you have created some classes with the names such as Pass and Detail in an application so that the purpose of a class is identified by its name. Now, if you want to use the classes with the same names, then you have to create another application to do so. However, in Java, you cannot create multiple classes with the same name.

To resolve class name collision, you can put all the related classes in a package. Now, going back to the example, the Pass and Detail classes that you created can be put in a package and in this way use the same names to create other classes outside the package.

In Java, packages are divided into the following two broad categories:

- User-defined packages
- Built-in packages

The Java Application Programming Interface (API) consists of many classes and interfaces, which are arranged in the form of packages according to their functionality. These are the built-in packages of the Java API. Apart from using these built-in packages, you can also create your own packages in an application. Let's discuss how to create packages in the following section.

Creating a Package

A package created by a user to perform some specific tasks is known as a user-defined package. User-defined packages can also be imported and used in the same way as built-in packages. To create a package, you have to use the `package` keyword. The general syntax to create a package is:

```
package <pack_name>; // statement to create a package
```

In the preceding syntax, `pack_name` is any logical name that you want to assign to your package. A package statement must be the first statement in a Java source file. Listing 10 shows how to create a package (you can find the `PackageDemo.java` file on the CD in the code\chapter6 folder):

▶ Listing 10: Creating a Package

```
package kogent; // package creation
public class PackageDemo // class in a package
{
  int x[];
  public PackageDemo(int a[]) // constructor having array as parameter
  {
```

```
        for (int i=0; i<a.length; i++)
        {
            x = a; // Assigning array elements to another array
        }
    }
    public void arr () // method for bubble sort
    {
        int temp=0, j, k;
        for (j=0; j<x.length; j++)
        {
            for(k=0; k<x.length-1; k++)
            {
                if (x[k]>x[k+1])
                {
                    temp=x[k+1];
                    x[k+1]= x[k];
                    x[k]=temp;
                }
            }
        }
        System.out.println("Elements after sorting are:");
        for (j=0; j<x.length; j++)
        {
            System.out.println(x[j]); // Printing sorted array elements
            on command prompt
        }
    }
    public static void main (String args[])
    {
        int m[]={5,3,2}; // Decalring and initializing array
        PackageDemo val = new PackageDemo(m); // creating object of class
        and passing array
        val.arr(); // calling arr method
    }
}
```

In Listing 10, the `kogent` package is created, which contains the `PackageDemo` class. This class is then used to perform the sorting operation on numbers, as shown in Listing 10, through the bubble sort algorithm. In the PackageDemo constructor, the `m` array is passed as a parameter because the values on which the sorting has to be performed are stored in an array.

The `arr()` method is then created, in which the code to perform the sorting on the array elements has been written. In the `main()` method, an array is first declared and initialized, followed by the creation of an object called `val`. The reference of the array is then passed as a parameter while creating the `val` object. Finally, the `arr()` method is invoked to perform the bubble sort on the array.

Now, let's learn how to compile and execute a program contained in a package. The process of compiling a program in a package is similar to the one adopted to compile a basic Java class, but includes more options.

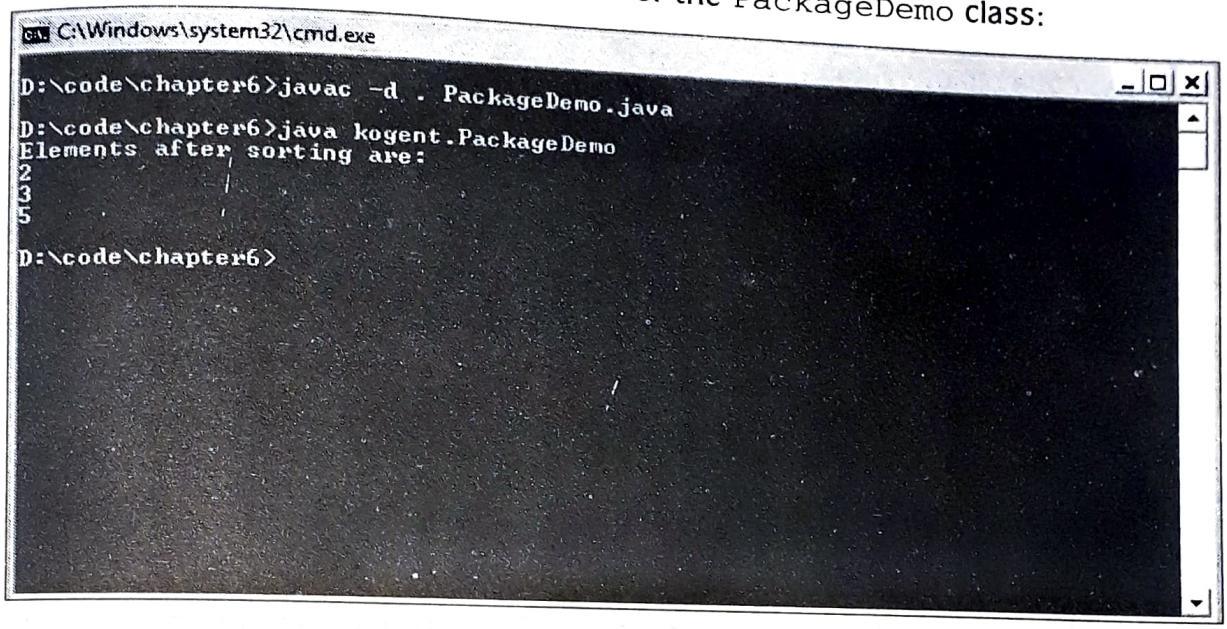
To compile a package, you need to instruct the compiler to create a package in the current directory. The following statement is used to create and compile a package in a program:

```
classpath>javac  -d . <class name>.java
```

In the preceding statement, class name is any logical name given to a class. After executing this statement, you need to run the program by referring the class name along with the package name. You can refer the class name by using '.' operator as shown in the following statement:

```
classpath>java <package name>.<class name>
```

The preceding statement executes the class, with its name specified after the package name. Figure 13 shows the compilation and execution of the `PackageDemo` class:



▲ **Figure 13: Output of the PackageDemo Program**

In Listing 10, a single program to perform a sorting operation is created. You can create more such programs and put them in the same package. In this way, you can create separate packages to assemble related Java programs.

TEST YOUR KNOWLEDGE

Q7. Write a program to create the com.kogent package.

Ans.

```
package com.kogent;
public class TestYourKnowledge7
{
 public static void main(String ar[])
 {
     System.out.println("Package com.kogent created");
 }
}
```

Execution:

```
D:\code\chapter6>javac -d . TestYourKnowledge7.java
D:\code\chapter6>java com.kogent.TestYourKnowledge7
```

Output:

```
Package com.kogent created
```

Now, let's learn how to import packages.

■ Importing Packages

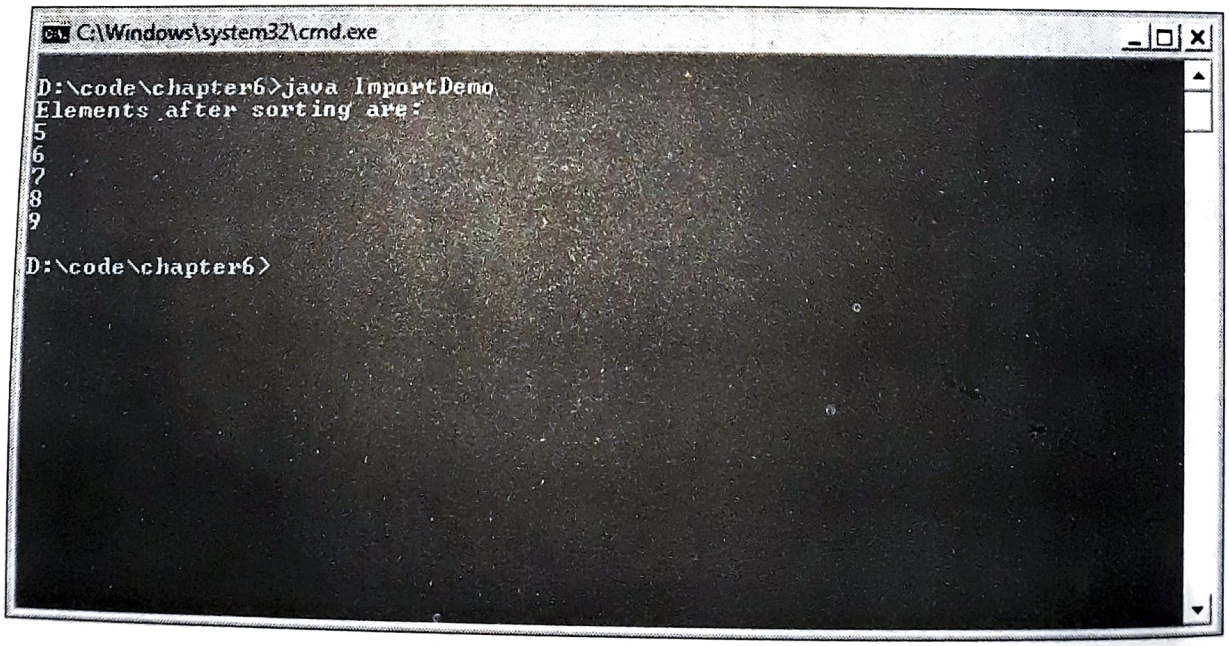
Importing packages in an application allows you to take the advantages of the classes stored in packages. You can import as many packages as you need to use in an application. You can also import user-defined packages. You can take the benefit of any package by using the import statement with the package name in a Java program. The import statement must come after the package specification and before the class definition. Listing 11 shows how to import packages by using the import statement (you can find the `ImportDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 11: Using the Import Statement**

```
import kogent.PackageDemo; // importing package
class ImportDemo
{
 public static void main(String args[])
 {
     int m[]={8,5,7,6,9};
     PackageDemo obj = new PackageDemo(m);
     obj.arr();
 }
}
```

In Listing 11, the `kogent` package created in Listing 10 has been imported. The `ImportDemo` class is then created, in which an array is first declared and initialized, and then an object of the `PackageDemo` class, which is imported from the `kogent` package, is created. Finally, the `arr()` method is invoked to perform sorting on the array elements.

In Listing 11, some elements in the array on which sorting has to be done have been passed at the time of object creation. However, the code to perform sorting has not yet been written in this listing because the code is already written in the package that we have imported. By importing packages, we can simplify our work by using the code written in the programs stored in the packages. Figure 14 shows the output of Listing 11:



▲ **Figure 14: Output of the ImportDemo Program**

In Figure 14, you can see that the elements stored in the array are successfully sorted. In Listing 11, only a single class of the kogent package is imported.

TEST YOUR KNOWLEDGE

Q8. Write a program to show use of the import statement.

Ans.

```
import java.util.Date;
public class TestYourKnowledge8
{
  public static void main(String ar[])
  {
      Date dt=new Date();
      System.out.println("Today is "+dt );
  }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge8.java
D:\code\chapter6>java TestYourKnowledge8
```

Output:

```
Today is Tue Nov 09 18:29:20 IST 20010
```

You can import more than one class of a package by writing an explicit import statement for each class or use the * notation, as shown in following code snippet:

```
import kogent.*;
```

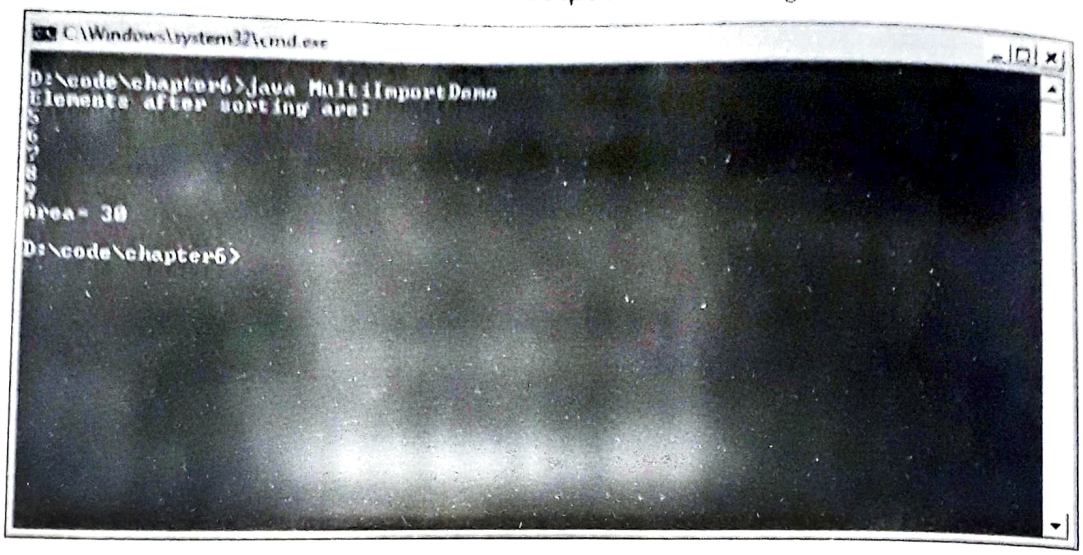
In the preceding code snippet, the * notation is used to import all the classes and interfaces of the kogent package. The use of the * notation avoids the repetitive use of import statements in the code. Listing 12 shows how to use the * notation for importing all the classes and interfaces of a package (you can find the `MultiImportDemo.java` file in CD in code\chapter6 folder):

▶ **Listing 12: Use of the * Notation to Import Multiple Classes and Interfaces**

```
import kogent.*;
class MultiImportDemo {
  public static void main(String args[]) {
      int m[]={8,5,7,6,9};
      kogent.PackageDemo obj = new kogent.PackageDemo(m);
      kogent.PackageDemo1 obj1 = new kogent.PackageDemo1();
      obj.arr();
      obj1.area(5,6);
  }
}
```

In Listing 12, the MultiImportDemo class is created to show the use of the * notation to import the entire package. Two classes, PackageDemo and PackageDemo1, are available in

kogent package. The arr() and area() methods of these classes are used in the MultiImportDemo class. Figure 15 shows the output of the Listing 12:



▲ Figure 15: Output of the MultiImportDemo Program

The PackageDemo1.java file is available on the CD in the code\chapter6 folder.

The result of invoking the arr() and area() methods is displayed in Figure 15. Although these methods are not available in the `MultiImportDemo` class, they are imported from the classes of the kogent package.

The classes and its members in a package can be protected from being accessed outside the package by using access specifiers available in Java. We discuss access protection in the next section.

Demonstrating Access Protection

As discussed in *Working with Java Members and Control Flow Statements, Chapter 2* Java has public, protected, private and default access specifiers. These access specifiers are used while declaring classes and their members. Depending on the access specifiers used, the classes and their members are accessible in other packages. Table 1 lists the access specifiers and the accessibility of the class members:

Table 1: Access Specifiers and the Accessiblity of Class Members

	Public	Protected	Private	Default
Same class	Yes	Yes	Yes	Yes
Same package subclass	Yes	Yes	No	Yes
Same package Non-subclass	Yes	Yes	No	Yes
Different package Subclass	Yes	Yes	No	No
Different package Non-subclass	Yes	No	No	No

As shown in Table 1, a class or class member declared with the public access specifier can be accessed from anywhere. When a class member is declared without an access specifier, the class member is accessible from within the same class and its subclass, from within other

classes in the same package, but not from outside the package it is declared. When a class member is declared with the protected access specifier, it is accessible from within the same class and its subclass, from within the other classes in the same package, and from the subclass outside the package. Private class members are accessible only from within the same class. Listing 13 declares a superclass within a package (you can also find the `SupClass.java` file on the CD in the code\chapter6 folder):

▶ **Listing 13: Declaring a Superclass in a Package**

```
package kogent;
public class SupClass {
  int defaultnum=5;
  private int privatenum=10;
  protected int protectednum=15;
  public int publicnum=20;
  public SupClass()
  {
      System.out.println("Simple Constructor");
      System.out.println("Default Number :"+defaultnum);
      System.out.println("Private Number :"+privatenum);
      System.out.println("Protected Number :"+protectednum);
      System.out.println("Public Number :"+publicnum);
  }
}
```

In Listing 13, the `SupClass` class is created, which declares a few member variables with all the access specifiers and initializes them. Listing 14 declares the `SubClass` subclass by extending the `SupClass` class (you can find the `SubClass.java` file on the CD in the code\chapter6 folder):

▶ **Listing 14: Declaring a Subclass in the Same Package**

```
package kogent;
class SubClass extends SupClass {
  public SubClass()
  {
      System.out.println("Constructor of Subclass");
      System.out.println("Default Number :"+defaultnum);
      //System.out.println("Private Number :"+privatenum);
      System.out.println("Protected Number :"+protectednum);
      System.out.println("Public :"+publicnum);
  }
}
```

In Listing 14, the `SubClass` class is created, which extends the `SupClass` class. The members of the `SupClass` class are accessed and displayed inside the constructor of the `SubClass` class. Note the line marked as comment, which is used to access the private member of the `SupClass` class. Not marking this line as comment causes a compilation error, as the private members of a class are not accessible outside that class. Listing 15 declares a simple class without using any access specifier, within the same package (you can also find the `ClassDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 15: Simple Class Declaration in the Same Package**

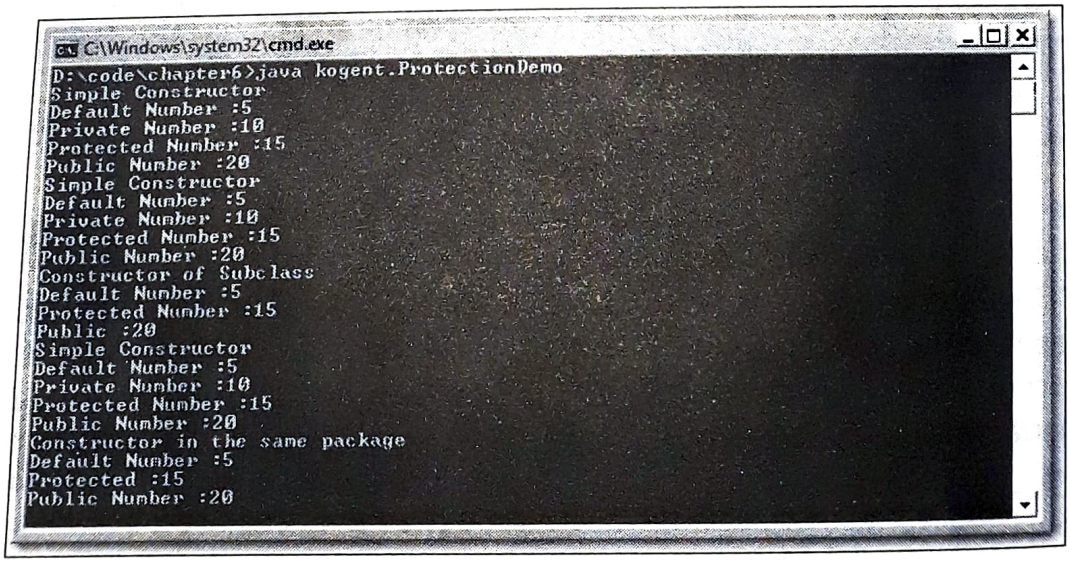
```
package kogent;
class ClassDemo
{
 ClassDemo()
 {
     SupClass obj=new SupClass();
     System.out.println("Constructor in the same package");
     System.out.println("Default Number :"+obj.defaultnum);
     System.out.println("Private Number :"+privatenum);
     //System.out.println("Protected :"+obj.protectednum);
     System.out.println("Protected :"+obj.publicnum);
 }
}
```

In Listing 15, the `ClassDemo` class is created, which is in the same package, `kogent`. It does not use any access specifier and is therefore considered with the default access specifier. The `ClassDemo` class accesses the members of the `SupClass` class. Similar to Listing 14, the private member being accessed (privatenum) is marked as a comment as it is not accessible in this class. Listing 16 creates a class to demonstrate the use of all the classes created in Listings 13, 14, and 15 (you can find the `ProtectionDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 16: Declaring the Main Class in the Same Package**

```
package kogent;
public  class ProtectionDemo {
 public static void main(String args[]) {
     SupClass obj=new SupClass();
     SubClass obj1=new SubClass();
     ClassDemo obj2 =new ClassDemo();
 }
}
```

In Listing 16, the ProtectionDemo class is created, which instantiates the objects of the `SupClass`, `SubClass`, and `ClassDemo` classes. When you compile and execute the code of Listings 13, 14, 15, and 16, the output is displayed, as shown in Figure 16:



▲ **Figure 16: Output of the ProtectionDemo Program**

As shown in Figure 16, the public, protected and default data members of the `SupClass` class are accessible by all the classes of the `kogent` package.

Java API contains several built-in classes and interfaces, which you can use in a program. All these classes and interfaces are arranged in packages called built-in packages. Let's discuss some important built-in packages of the Java API in the following section.

6.9 Defining Java API Packages

Similar to other programming languages, Java also provides some built-in functionalities that you can use in your programs. These built-in functionalities are coded in the form of classes and interfaces. These classes and interfaces are grouped in packages according to their purpose. Table 2 lists the important packages available in the Java API:

Table 2: Packages Available in Java API

Package Name	Description
java.lang	Contains essential Java classes, including Number, String, Object, Compiler, Runtime, Security, and Thread. This is the only package that is automatically imported into every Java program.
java.io	Provides classes to manage input and output streams to read and write data, respectively, to files, strings, and other sources.
java.util	Provides miscellaneous utility classes, including generic data structures, bit sets, Time, Date, string manipulation classes, random number generation classes, system properties classes, notification classes, and enumeration of data structures classes.
java.net	Contains the classes used to implement network applications. It provides classes for network support, including URLs, TCP and UDP sockets, IP addresses, and a binary-to-text converter.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics, images, windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields. AWT stands for Abstract Window Toolkit.
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context. It enables the creation of applets through the Applet class. It also provides several interfaces that connect an applet to its document and to resources for playing audio.

The java.lang and java.util packages are the most commonly used packages while writing Java programs. Let's discuss these packages in detail in the following sections.

▪ The java.lang Package

The java.lang package is the most important and basic package of the Java API. It provides classes to perform basic functions in a program. You do not need to import this package as it is automatically imported in your program. Table 3 lists the classes available in the java.lang package:

Table 3: Classes of the java.lang Package

Class Name	Description
Boolean	A wrapper class that wraps a value of primitive type boolean in an object.
Byte	A wrapper class that wraps a value of primitive type byte in an object.
Character	A wrapper class that wraps a value of primitive type char in an object.
Class	Represents the instances of the classes and interfaces in a running Java application.
ClassLoader	An instance of the ClassLoader class is responsible for loading classes during runtime.
Compiler	Provides support to Java-to-native-code compilers and related services.
Double	A wrapper class that wraps a value of primitive type double in an object.
Float	A wrapper class that wraps a value of primitive type float in an object.
Integer	A wrapper class that wraps a value of primitive type int in an object.
Long	A wrapper class that wraps a value of primitive type long in an object.
Math	Provides methods to perform basic numeric operations, such as the elementary exponential, logarithm, square root, and trigonometric functions.
Number	Provides methods to work with different types of numbers in a Java program. It is an abstract class and is the superclass of the BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short classes.
Object	The root of the class hierarchy. All classes in Java extend the Object class implicitly.
Package	Provides version information about the implementation and specifications of a Java package.
Runtime	An instance of the Runtime class allows an application to interface with the environment in which the application is running.
SecurityManager	Allows applications to implement security policies.
Short	A wrapper class that wraps a value of primitive type short in an object.
String	Represents character strings. String is not a primitive data type in Java; rather, the String class is used for string manipulation.
StringBuffer	Implements a mutable sequence of characters.

Table 3: Classes of the java.lang Package

Class Name	Description
System	Provides standard input, output, and error output streams; access to externally defined properties of the system, and environment variables such as Temp and Username; a means to load files and libraries; and a utility method for quickly copying a portion of an array.
Thread	Creates and uses threads in a Java program.
Throwable	The superclass of all errors and exceptions in the Java language.
Void	Holds a reference to the Class object representing the Java void keyword. The Void class is an uninstantiable placeholder class.

Table 3 lists important classes of the java.lang package. Apart from these classes, the java.lang package also contains some interface, exception, and error classes. Let's discuss the most commonly used classes of the java.lang package in detail in the following sections.

The Object Class

The Object class is the base class of all the classes created in Java. Classes created in Java programs automatically extend the Object class. Since it acts as the superclass of all the Java classes, a reference of the Object class can be used to refer any object in a Java program. Table 4 lists noteworthy methods of the Object class:

Table 4: Noteworthy Methods of the Object Class

Method Name	Description
protected Object clone()	Creates and returns a copy of the current object.
boolean equals(Object obj)	Determines whether or not some other object is equal to the current object.
protected void finalize()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class getClass()	Returns the runtime class of an object.
int hashCode()	Returns a hash code value for an object.
void notify()	Wakes a single thread that is waiting on the current object's monitor.
void notifyAll()	Wakes all the threads that are waiting on the current object lock.
String toString()	Returns a string representation of the object that is invoking this method.
void wait()	Causes the current thread to wait until another thread invokes the notify()or notifyAll() method for the current object.
void wait(long time)	Causes the current thread to wait until another thread invokes the notify() or notifyAll() method for the current object, or until the specified amount of time has elapsed.

Table 4: Noteworthy Methods of the Object Class

Method Name	Description
void wait(long time, int nano)	Causes the current thread to wait until another thread invokes the notify() or notifyAll() method for the current object, or until another thread interrupts the current thread.

The getClass(), notify() notifyAll(), and wait() methods listed in Table 4 are final methods and cannot be overridden. Listing 17 shows an example of using the hashCode() method (you can find the HashCodeDemo.java file on the CD in code\chapter6 folder):

▶ **Listing 17: Using the hashCode() Method**

```
public class HashCodeDemo
{
  public static void main(String[] args)
  {
      HashCodeDemo obj = new HashCodeDemo();
      int hashcode = obj.hashCode();
      System.out.println("The hash code of obj is " +hashcode);
  }
}
```

In Listing 17, the HashCodeDemo class is created, which uses the hashCode() method to get the hash code for an object. Figure 17 shows the output of Listing 17:

▲ Figure 17: Output of the HashCodeDemo Program

In Figure 17, the hashCode() method retrieves the hash code of the obj object.

TEST YOUR KNOWLEDGE

Q9. Write a program to retrieve the hash code of an object.

Ans.

```
public class TestYourKnowledge9
{
  public static void main(String[] args)
  {
      TestYourKnowledge9 obj = new TestYourKnowledge9();
      int hashcode = obj.hashCode();
      System.out.println("The hash code of obj is " +hashcode);
```

```
    }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge9.java
D:\code\chapter6>java TestYourKnowledge9
```

Output:

```
The hash code of obj is 4072869
```

Next, let's discuss another important class in the java.lang package, that is the Class class.

The Class Class

The Class class is used to represent instances of classes and interfaces in a running Java application. It provides methods that support runtime processing of an object's class, and interface information. Table 5 lists noteworthy methods of the Class class:

Table 5: Noteworthy Methods of the Class Class

Method Name	Description
String getName()	Retrieves the name of the entity (class, interface, array class, primitive type, or void) represented by the current Class object, as a String.
String toString()	Changes the current object into a string.
static Class forName(String className)	Retrieves the Class class object associated with a class or interface with the given string name.
Class getSuperclass()	Retrieves the class representing the superclass of the entity (class, interface, primitive type or void) represented by the current class.
Class[] getInterfaces()	Determines the interfaces implemented by the class or interface represented by the current object.
boolean isInterface()	Determines if the specified Class object represents an interface type.
T newInstance()	Returns a new instance of the class represented by the current Class object.
ClassLoader getClassLoader()	Retrieves the class loader for the current class.

TEST YOUR KNOWLEDGE

Q10. Write a program to get the name of the current thread.

Ans.

```
public class TestYourKnowledge10 implements Runnable {
  static Thread t1;
  public void run() {
     System.out.println(Thread.currentThread().getName());
  }
  public static void main(String args[]) {
```

```
        TestYourKnowledge10 t=new TestYourKnowledge10();
        t1=new Thread(t);
        t1.start();
    }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge10.java
D:\code\chapter6>java TestYourKnowledge10
```

Output:

```
Thread-0
```

The Runtime Class

The Runtime class provides access to the Java runtime system. The instance of the Runtime class allows an application to interact with the environment in which the application is running. Table 6 lists noteworthy methods of the Runtime class:

Table 6: Noteworthy Methods of the Runtime Class

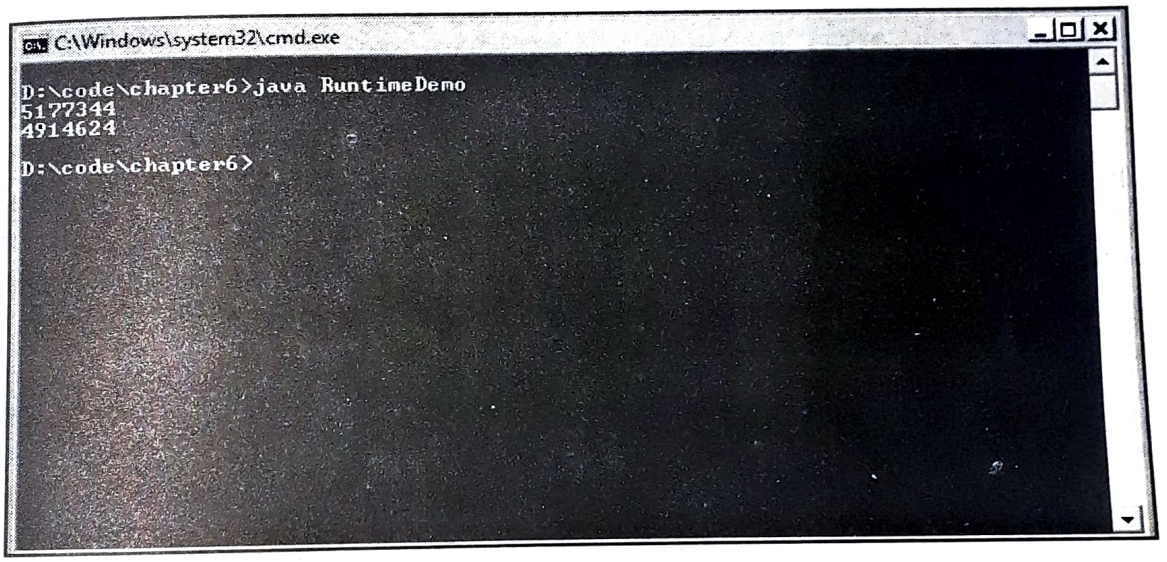
Method Name	Description
static Runtime getRuntime()	Returns the runtime object associated with the current Java application.
Process exec(String command)	Executes the specified string command in a separate process.
long freeMemory()	Returns the amount of free memory in the JVM.
InputStream getLocalizedInputStream(InputStream in)	This method is deprecated. From JDK version 1.1 onwards, the preferred way to translate a byte stream in local encoding into a character stream in Unicode is through the InputStreamReader and BufferedReader classes.
OutputStream getLocalizedOutputStream(OutputStream out)	This method is deprecated. From JDK version 1.1 onwards, the preferred way to translate a Unicode character stream into a byte stream in local encoding is through the OutputStreamWriter, BufferedWriter, and PrintWriter classes.
void runFinalization()	Runs the finalization methods of any objects pending finalization.
void traceInstructions(boolean on)	Instructs the JVM to print a detailed trace of each instruction as it is executed, provided the boolean parameter is true. The destination where the JVM displays the detail of the trace output is system-dependent. If the boolean argument is false, then this method causes the JVM to stop performing a detailed instruction trace.
void traceMethodCalls(boolean on)	Enables/disables tracing of method calls.

Most of the methods of the Runtime class are not used in application programming. However, a few of them are useful. Listing 18 shows an example of using the totalMemory() and freeMemory() methods (you can also find the `RuntimeDemo.java` file on the CD in the code\chapter6 folder):

▶ Listing 18: Using Methods of the Runtime Class

```
public class RuntimeDemo
{
  public static void main(String args[])
  {
      Runtime run = Runtime.getRuntime();
      System.out.println(run.totalMemory());
      System.out.println(run.freeMemory());
  }
}
```

In Listing 18, the `RuntimeDemo` class is created, which instantiates the `Runtime` class to get the current runtime environment. The `totalMemory()` and `freeMemory()` methods are invoked by using the `run` instance of the `Runtime` class, which displays the output shown in Figure 18:



▲ **Figure 18: Output of the RuntimeDemo Program**

The `RuntimeDemo` class uses the `getRuntime()` method to get the current runtime environment. The free and the total memory is calculated by using the `freeMemory()` and `totalMemory()` methods, respectively.

TEST YOUR KNOWLEDGE

Q11. Write a program to get the total memory of a JVM.

Ans.

```
public class TestYourKnowledge11
{
  public static void main(String args[])
  {
      Runtime run = Runtime.getRuntime();
      System.out.println("Total Memory "+run.totalMemory());
```

```
    }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge11.java
D:\code\chapter6>java TestYourKnowledge11
```

Output:

```
Total Memory 5177344
```

You can even execute another application from a Java program by using the `exec()` method. Listing 19 shows an example of using the `exec()` method (you can also find the `ExecDemo.java` file on the CD in code\chapter6 folder):

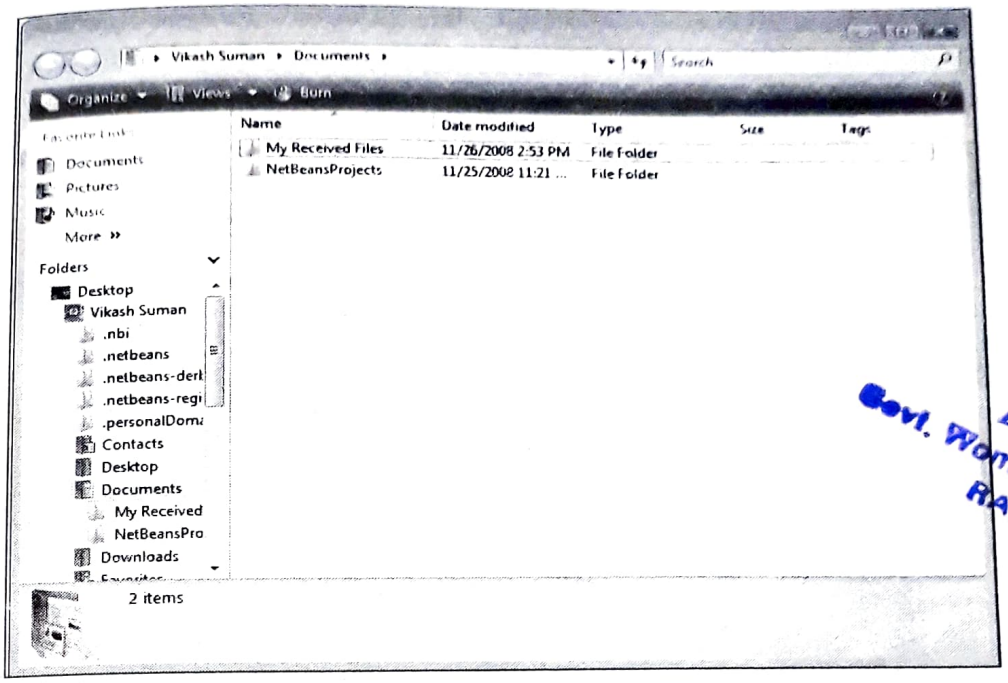
▶ **Listing 19: Using the exec() Method**

```
import java.io.*;
public class ExecDemo
{
  public static void main(String args[]) throws IOException
  {
      Runtime run = Runtime.getRuntime();
      run.exec("C:\\Windows\\Explorer.exe");
  }
}
```

In Listing 19, the `ExecDemo` class instantiates the `Runtime` class and retrieves the current runtime object by invoking the getRuntime() method. The exec() method is used to start Windows Explorer. Figure 19 shows execution of Listing 19:

▲ **Figure 19: Execution of the ExecDemo Program**

When you execute the program (Figure 19), the `Windows Explorer` window appears on your screen, as shown in Figure 20:



▲ **Figure 20: Displaying Windows Explorer as Output of the ExecDemo Program**

An important class of the `java.lang` package that allows you to work with different types of numeric values is the `Number` class. Let's therefore discuss the `Number` class in detail in the following section.

The Number Class

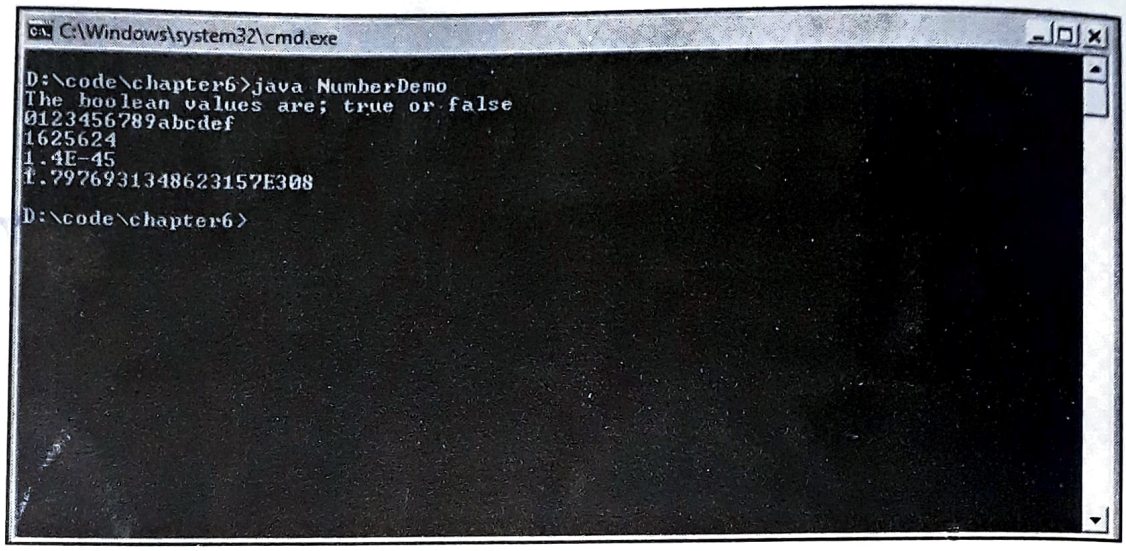
The `Number` class is an abstract class of the `java.lang` package. This class is inherited by the `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short` classes. The subclasses of the `Number` class are wrapper classes and are used to wrap primitive data types such as `int`, `long`, and `double`. Listing 20 shows an example of using methods of the `Number` class (you can also find the `NumberDemo.java` file on the CD in the code\chapter6 folder):

▶ **Listing 20: Using Methods of the Number Class**

```
public class NumberDemo
{
 public static void main(String ar[])
 {
    Boolean bool1 = new Boolean("TRUE");
    Boolean bool2 = new Boolean("FALSE");
    System.out.println("The boolean values are; "+bool1.toString()+" or
    "+bool2.toString());
    for(int j=0;j<16;++j)
    System.out.print(Character.forDigit(j,16));
    System.out.println();
    Integer intgr = new Integer(Integer.parseInt("AB",16));
    Long lng = new Long(Long.parseLong("ABC",16));
    long lngval=lng.longValue()*intgr.longValue();
    System.out.println(Long.toString(lngval,8));
    System.out.println(Float.MIN_VALUE);
    System.out.println(Double.MAX_VALUE);
```

```
    }
}
```

In Listing 20, the `NumberDemo` class uses different methods and wrapper classes. The `Boolean`, `Integer` and `Long` classes are used to wrap their respective primitive data types. Figure 21 shows the output of Listing 20:



▲ **Figure 21: Output of the NumberDemo Program**

The primitive data types are wrapped by their respective wrapper classes and their values displayed, as shown in Figure 20.

By now, we have discussed all the important classes of the `java.lang` package in detail. In the next section, we take up another important package: `java.util`.

■ The java.util Package

The `java.util` package contains general utility classes. It contains the collections framework, legacy collection classes, event models, date and time facilities, classes that support the internationalization feature, and miscellaneous utility classes, such as the string tokenizer, random-number generator, and bit array. Let's discuss some of the important classes of the `java.util` package in detail in the following sections.

➡ The Hashtable Class

The Hashtable class implements a hashtable table, which maps the keys to the values associated with an object. Objects are used as key value pairs while creating the hashtable table. To successfully store and retrieve objects from a hashtable table, the objects used as keys must implement the `hashCode()` and the `equals()` methods. The following code snippet shows how to create a hashtable table of numbers:

```
Hashtable numtable = new Hashtable();
numtable.put("First", new Integer(1));
numtable.put("Second", new Integer(2));
numtable.put("Third", new Integer(3));
numtable.put("Fourth", new Integer(4));
numtable.put("Fifth", new Integer(5));
```

In the preceding code snippet, the `numtable` hashtable table is created, and five elements are added to it by using the `put()` method. To retrieve a number from this hashtable table, use the following code snippet:

```
Integer intgr= (Integer)numtable.get("Fifth");
if (n != null)
{
  system.out.println("Fifth element = " + intgr);
}
```

In the preceding code snippet, the `get()` method is used to retrieve the fifth value from the hashtable table. The `Hashtable` class contains overloaded constructors, which can be used to create a hashtable table. Table 7 lists the constructors of the `Hashtable` class:

Table 7: Constructors of the Hashtable Class

Constructor	Description
Hashtable()	Creates a new, empty hashtable table with a default initial capacity of 11 (length of the array used to implement the hashtable table) and a load factor of 0.75 (ratio of the number of key/value pairs in the hashtable table to the array length).
Hashtable(int initialCapacity)	Creates a new, empty hashtable table with the specified initial capacity and default a load factor of 0.75.
Hashtable(int initialCapacity, float loadFactor)	Constructs a new, empty hashtable table with the specified initial capacity and load factor.
Hashtable(Map t)	Constructs a new hashtable table with the same mappings as the given map as a method parameter.

The `Hashtable` class contains methods that are used to work with a hashtable table. Table 8 lists noteworthy methods of the `Hashtable` class:

Table 8: Noteworthy Methods of the Hashtable Class

Method Name	Description
int size()	Retrieves the number of keys in a hashtable table used to invoke the table.
public boolean isEmpty()	Returns true or false, depending on whether or not the hashtable table contains the mapping for keys to their corresponding values.
public synchronized Enumeration keys()	Retrieves the keys of a hashtable table in an enumeration.
public synchronized boolean contains(Object value)	Determines whether or not a key maps to the specified value in a hashtable table.
public synchronized boolean containsKey(Object key)	Determines whether or not the specified object is a key in a hashtable table.
public synchronized Object get(Object key)	Retrieves the value to which the specified key is mapped in a hashtable table.

Table 8: Noteworthy Methods of the Hashtable Class

Method Name	Description
protected void rehash()	Increases the capacity of a hashtable table and reorganizes it internally to accommodate and access its entries more efficiently.
public synchronized Object put(Object key, Object value)	Inserts the specified key to the specified value in the current hashtable table.
public synchronized Object remove(Object key)	Removes the key and its corresponding value from the current hashtable table if the key is available.
public synchronized clear()	Removes all the entries in a hashtable table so that it contains no keys.
public synchronized Object clone()	Creates a shallow copy (a shallow copy simply copies chunks of memory from one location to another. It does not copy the object semantics) of a hashtable table.
public synchronized String toString()	Returns a string representation of a Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space).

Next, we describe the `Date` class of the `java.util` package.

➡ The Date Class

The `Date` class is a wrapper for representing the date in a platform-independent manner. The `Date` class has overloaded constructors and methods to manipulate the date in different formats in an application. Table 9 lists the constructors of the `Date` class:

Table 9: Constructors of the Date Class

Constructor	Description
public Date()	Creates a Date object and initializes it to represent the time at which this object was allocated, measured to the nearest millisecond.
public Date(int year, int month, int date)	This constructor is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.set(year + 1900, month, date) or GregorianCalendar(year + 1900, month, date) constructor.
public Date(int year, int month, int date, int hrs, int min)	This constructor is deprecated. From JDK version 1.1 onwards, it is, replaced by Calendar.set(year + 1900, month, date, hrs, min) or GregorianCalendar(year + 1900, month, date, hrs, min) constructor.
public Date(int year, int month, int date, int hrs, int min, int sec)	This constructor is deprecated. From JDK version 1.1 onwards, it is replaced by Calendar.set(year + 1900, month, date, hrs, min, sec) or GregorianCalendar(year + 1900, month, date, hrs, min, sec) constructor.
public Date(long date)	Creates a Date object and initializes it to represent the specified number of milliseconds since the standard base time, known as the epoch, that is 1, 1970, 00:00:00 GMT.

Table 9: Constructors of the Date Class

Constructor	Description
public Date(String s)	This constructor is deprecated. Form JDK version 1.1 onwards, it is replaced by DateFormat.parse(String s) constructor.

A few of the constructors listed in Table 9 are deprecated. Alternative ways are provided for deprecated constructors to instantiate the `Date` class.

The following code snippet is used to create a date object, which represents the current date and time:

```
Date dt=new Date();
System.out.println("Today's Date = "+ dt);
```

In the preceding code snippet, the `dt` instance of the `Date` class is created to represent the current date and time. The `Date` class provides several methods to manipulate the date in a platform-independent manner. Table 10 lists the methods of the `Date` class:

Table 10: Methods of the Date Class

Method Name	Description
public static long UTC(int year, int month, int date, int hrs, int min, int sec)	This method is deprecated. From JDK version 1.1 onwards, it is replaced by Calendar.set(year + 1900, month, date, hrs, min, sec) or GregorianCalendar(year + 1900, month, date, hrs, min, sec) method, using a UTC TimeZone, followed by the Calendar.getTime().getTime() method.
public static long parse(String s)	This method is deprecated. Form JDK version 1.1 onwards, it is replaced by the DateFormat.parse(String s) method.
public int getYear()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.get(Calendar.YEAR) – 1900 method.
public void setYear(int year)	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.set(Calendar.YEAR, year + 1900) method.
public void setMonth(int month)	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.set(Calendar.MONTH, int month) method.
public int getMonth()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.get(Calendar.MONTH) method.
public int getDate()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.get(Calendar.DAY_OF_MONTH) method.
public void setDate(int date)	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the

Table 10: Methods of the Date Class

Method Name	Description
	Calendar.set(Calendar.DAY_OF_MONTH, int date) method.
public int getDay()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.get(Calendar.DAY_OF_WEEK) method.
public int getHours()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.get(Calendar.HOUR_OF_DAY) method.
public void setHours(int hours)	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.set(Calendar.HOUR_OF_DAY, int hours).
public int getMinutes()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.get(Calendar.MINUTE) method.
public void setMinutes(int minutes)	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.set(Calendar.MINUTE, int minutes) method.
public int getSeconds()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.get(Calendar.SECOND) method.
public void setSeconds(int seconds)	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the Calendar.set(Calendar.SECOND, int seconds) method.
public void setTime(long time)	Sets the current Date object to represent a point in time, that is, time in milliseconds after January 1, 1970 00:00:00 GMT.
public long getTime()	Retrieves the number of milliseconds since January 1, 1970, 00:00:00 GMT, represented by current Date object.
public boolean before(Date whn)	Determines if the current date is before the specified date.
public boolean equals(Object obj)	Compares two dates to check whether or not they are the same.
public int hashCode()	Retrieves the hash code value for the current object.
public String toString()	Converts a Date object to a String of the form: dow mon dd hh:mm:ss zzz yyyy where, ▪ dow is the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat). ▪ mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec). ▪ dd is the day of the month (01 through 31), as two

Table 10: Methods of the Date Class

Method Name	Description
	decimal digits. ▪ hh is the hour of the day (00 through 23), as two decimal digits. ▪ mm is the minute within the hour (00 through 59), as two decimal digits. ▪ ss is the second within the minute (00 through 59, as two decimal digits. ▪ zzz is the time zone (and may reflect daylight saving time). Standard time zone abbreviations include those recognized by the parse() method. If time zone information is not available, then zzz is empty, that is, it consists of no characters at all. ▪ yyyy is the year, as four decimal digits.
public String toLocaleString()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the DateFormat.format(Date date) method.
public String toGMTString()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the DateFormat.format(Date date) method, using a GMT TimeZone method.
public int getTimezoneOffset()	This method is deprecated. From JDK version 1.1 onwards, it is replaced by the (Calendar.get(Calendar.ZONE_OFFSET) + Calendar.get(Calendar.DST_OFFSET)) / (60 * 1000) method.

Next, we describe the `Random` of the `java.util` package.

The Random Class

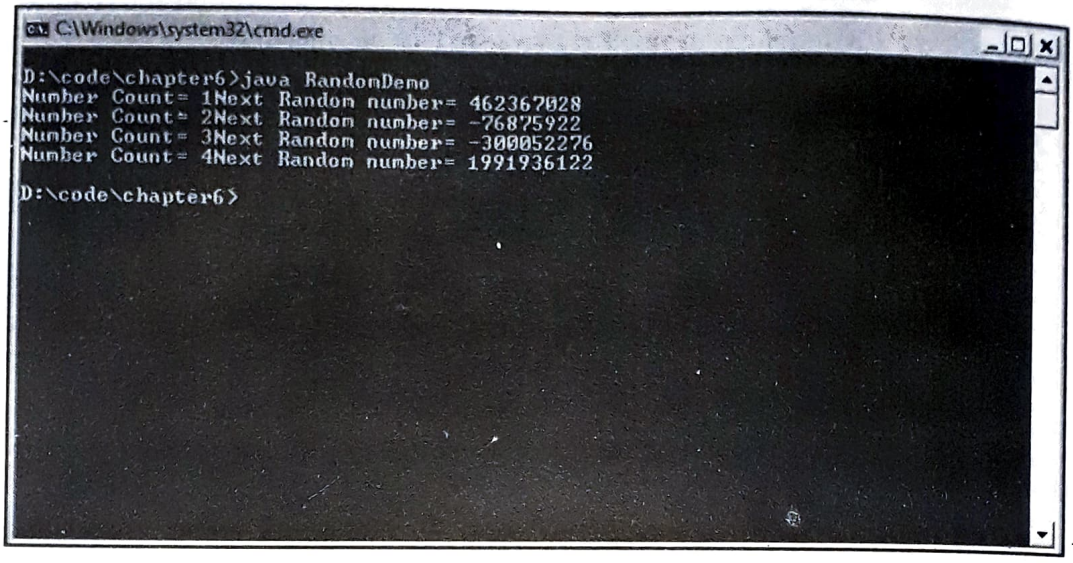
You sometimes need to generate numbers in random order in a Java application. An instance of the `java.util.Random` class is used to generate a stream of random numbers. Listing 21 shows an example of generating random numbers (you can also find the `RandomDemo.java` file on the CD in the code\chapter6 folder):

▶ Listing 21: Creating Random Numbers

```
import java.util.Random;
public class RandomDemo
{
  static int cnt;
  static int number;
  static Random rndnum;
  public static void main(String ar[])
  {
      rndnum=new Random();
      for(cnt=1;cnt<5;cnt++)
      {
          number=rndnum.nextInt();
```

```
            System.out.print("Number Count= "+ cnt);
            System.out.println("Next Random number= "+ number);
        }
    }
}
```

In Listing 21, the RandomDemo class generates four random numbers. The nextInt() method is used to retrieve these random numbers. This method returns the next random number as an integer. Figure 22 shows the output of Listing 21:



▲ Figure 22: Output of the RandomDemo Program

Figure 22 shows the four generated random numbers. Apart from the nextInt() method, the Random class also contains other methods and constructors. Table 11 lists the constructors of the Random class:

Table 11: Constructors of the Random Class

Constructor	Description
public Random()	Constructs a new random number generator.
public Random(long seed)	Constructs a new random number generator by using a specified seed (an integer used to set the starting point for generating a series of random numbers).

Table 12 lists the methods of the Random class:

Table 12: Methods of the Random Class

Method Name	Description
protected int next(int bits)	Creates the next random number.
public boolean nextBoolean()	Retrieves the next boolean random number from the current random number generator's sequence.
public void nextBytes(byte[] bytes)	Creates random bytes and places them into the specified byte array.
public double nextDouble()	Retrieves the next double random number between 0.0 and 1.0, from the current random number generator's sequence.

Table 12: Methods of the Random Class

Method Name	Description
public float nextFloat()	Retrieves the next float random number between 0.0 and 1.0, from the current random number generator's sequence.
public double nextGaussian()	Retrieves a Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from the current random number generator's sequence.
public int nextInt()	Retrieves the next int random number distributed value from the current random number generator's sequence.
public int nextInt(int n)	Retrieves an int random number between 0 (inclusive) and the specified value (exclusive), drawn from the current random number generator's sequence.
public long nextLong()	Retrieves the next long random number from this random number generator's sequence.
public void setSeed(long seed)	Sets a seed of the current random number generator by using the specified seed.

TEST YOUR KNOWLEDGE

Q12. Write a program to generate three random numbers.

Ans.

```
import java.util.Random;
public class TestYourKnowledge12
{
  static int count;
  static int number;
  static Random rndnum;
  public static void main(String ar[])
  {
      rndnum=new Random();
      for(count=1;count<=3;count++)
      {
          number=rndnum.nextInt();
          System.out.print("Number Count= "+ count);
          System.out.println("Next Random number= "+ number);
      }
  }
}
```

Execution:

```
D:\code\chapter6>javac TestYourKnowledge11.java
D:\code\chapter6>java TestYourKnowledge11
```

Output:

```
Number Count= 1Next Random number= -1300001347
Number Count= 2Next Random number= 1025234314
Number Count= 3Next Random number= 769704028
```

We now describe to the `Calendar` class, which is another important class of the `java.util` package.

The Calendar Class

The Calendar class is an abstract class of the `java.util` package. It provides the methods to convert a specific instant in time into a set of calendar fields, such as YEAR, MONTH, DAY_OF_MONTH, and HOUR. It allows you to manipulate the fields, such as getting a date of the next week. A Calendar object can produce all the calendar field values needed to implement date-time formatting for a particular language and calendar style.

The Calendar class has two modes to interpret calendar fields, lenient and non-lenient. When a calendar is in the lenient mode, it accepts a wider range of calendar field values than it produces, whereas when the calendar is in the non-lenient mode, it throws an exception if there is any inconsistency in the calendar fields.

The Calendar class provides constructors and methods to work with the date and time. Table 13 lists the constructors of the Calendar class:

Table 13: Constructors of the Calendar Class

Constructor	Description
protected Calendar()	Creates a calendar with the default time zone and locale.
protected Calendar(TimeZone zone, Locale aLocale)	Creates a calendar with the specified time zone and locale.

Table 14 lists the some important methods of the Calendar class:

Table 14: Methods of the Calendar Class

Method Name	Description
public abstract void add(int field, int amount)	Adds or subtracts the specified amount of time to the specified calendar field.
public boolean after(Object when)	Determines whether or not the current Calendar object represents a time after the time represented by the specified object.
public boolean before(Object when)	Determines whether or not the current Calendar object represents a time before the time represented by the specified object.
public void clear()	Clears all the calendar field values and the time value of the current Calendar object.
public void clear(int field)	Clears the given calendar field value and time value of the current Calendar object.
public Object clone()	Creates and a copy of the current object.
public int compareTo(Calendar anotherCalendar)	Compares the time represented by two Calendar objects and returns the difference as an integer value.
protected void complete()	Fills and completes any unset fields in the calendar.
protected abstract void computeFields()	Changes the current millisecond time into calendar field values in the fields array.

protected abstract void computeTime()	Changes the current calendar field values in the fields array to the millisecond time format.
public boolean equals(Object obj)	Compares the current Calendar object to the specified object as method parameter.
public int get(int field)	Retrieves the value of the given calendar field.
public int getActualMaximum(int field)	Retrieves the maximum value that a specified calendar field can have.
public int getActualMinimum(int field)	Retrieves the minimum value that a specified calendar field can have.
public static Locale[] getAvailableLocales()	Retrieves an array of all locales for which the getInstance() method of the current class can return localized instances.
public int getFirstDayOfWeek()	Returns the first day of the week according to the locale. For example, SUNDAY in the U.S., MONDAY in France.
public abstract int getGreatestMinimum(int field)	Retrieves the highest minimum value for the given calendar field of the current Calendar instance.
public void setTime(Date date)	Sets the current calendar's time with the specified date.
public void setTimeInMillis(long millis)	Sets the current calendar's time from the given long value.
public void setTimeZone(TimeZone value)	Sets the time zone with the given time zone value.
public String toString()	Retrieves a string representation of the current calendar object.

With this, we come to the end of the chapter. Next is a summary to recap the main topics covered in this chapter.

Summary

In this chapter, you have learned about threads, their states, and setting their priority. It discusses the various states of a thread, that is, `new`, `runnable`, `running`, `blocked/waiting/sleeping`, and `dead`. In addition, this chapter describes how a thread makes a transition from one state to another by invoking various methods provided by the `Thread class`. It also discusses the behavior of the `start()`, `yield()`, `sleep()`, and `join()` methods and the thread states in which these methods can be called.

This chapter explains the concept of deadlock, the situations where deadlocks may result, and the reasons for their occurrence. It describes what happens when shared resources are accessed in a non-synchronized manner.

In addition, this chapter discusses the use of synchronized methods and blocks. It also discusses three methods, wait(), notify(), and notifyAll(), which control thread interaction. Apart from this, it describes the different types of exceptions that are thrown due to the invocation of the methods provided by the Thread and Object classes. It also focuses on two types of packages, user defined and built-in. It also discusses the important packages of the Java API.

Review Questions

Here let's answer the following types of questions:

- True or False
- Multiple Choice
- Short Answers
- Debugging

■ True/False Questions

State True or False for the following:

1. Threads can be created by either extending the Thread class or by implementing the Runnable interface. **True**
2. A single target object can be used only to create single threads in a program. **False**
3. The `start()` method can be invoked only on the threads in the new state. **True**
4. The `start()` method can be called only once in the life cycle of a thread. **True**
5. A thread moves to the running state once it has been started. **False**
6. Threads in the runnable state are selected by the thread scheduler for execution. **True**
7. Invocation of the `wait()`, `sleep()`, or `join()` method moves the current thread to the running state. **False**
8. The `join()` method is used to join one thread to another thread, which will wait until the calling thread terminates. **True**
9. A thread in the sleep state relinquishes its acquired locks. **False**
10. The `getPriority()` method is used to retrieve the priority of a thread. **True**
11. A package declaration should be the first statement of a Java source file. **True**
12. A single package can only be imported in a Java source file. **False**

■ Multiple Choice Questions

Answer the following multiple choice questions:

Q1. **Which of the following is incorrect about the wait() method?**

A. The wait() method can only be invoked by the thread that owns the lock on the current object.

B. The wait() method is defined in the Object class.

C. The Timeout parameter specified to the wait (long Timeout) method is in seconds.

D. The wait() method must be wrapped in a try/catch block as it throws the InterruptedException exception.

Ans. Option C is correct.

Q2. Which of the following statements are true?

A. The notifyAll() method is defined in the java.lang.Thread class.

B. The notify() and wait() methods can only be called from a synchronized method or block.

C. A single thread is notified by invoking the notifyAll() method.

D. The object that owns the lock on a thread can call the wait() method.

Ans. Option B is correct.

Q3. Which of the following is not a state of a thread?

A. New

B. Execute

C. Runnable

D. Running

Ans. Option B is correct.

Q4. Which keyword can protect a class member from being accesses from within the subclass?

A. public

B. protected

C. private

D. default

Ans. Option C is correct.

Q5. Which of the following is the method of the Runnable interface?

A. run()

B. wait()

C. start()

D. notify()

Ans. Option A is correct.

Q6. In which of the following state of a thread can the start() method be invoked?

A. New

B. Blocked

C. Runnable

D. Running

Ans. Option A is correct.

Q7. Which of the following packages is imported automatically in a Java program?

A. java.awt

B. java.net

C. java.lang

D. java.applet

Ans. Option C is correct.

Q8. Which of the following packages contains the Calendar class?

A. java.awt

B. java.net

C. java.lang

D. java.util

Ans. Option D is correct.

Q9. Which of the following methods is called when an object is garbage collected?

A. run()

B. finalize()

C. toString()

D. notify()

Ans. Option B is correct.

Q10. Which of the following methods is used to execute an external program from a Java application?

A. run()
B. execute()
C. invoke()
D. exec()

Ans. D

Q11. A package is a collection of ..... and .....

A. Classes and Interfaces
B. Codes and Variables

Ans. Option A is correct.

Q12. A..... member of a class can be accessed from anywhere in a Java application.

A. public
B. protected
C. private
D. default

Ans. Option A is correct.

■ Short Answer Questions

Answer the following short answer questions.

Q1. What is a thread?

Ans. A thread is the smallest unit of an executable code in a program. It helps you divide a process into multiple parts to speed up the process. A process is a program that executes in the memory as a single thread.

Q2. List the important uses of threads.

Ans. The important uses of threads are as follows:

- Threads are used in server-side programs to serve the needs of multiple clients on a network or the Internet. On the Internet, the server has to cater to the needs of thousands of clients at a time. Threads are used to handle multiple clients simultaneously.
- Threads are used to create games and animations. For example, you can use threads to show a picture in motion. In such a case, you have to set a thread such that it sleeps and is activated for specific periods of time, and the cycle is continued in this way. Threads can be used in this way to create animations.
- In many games, a user generally has to perform more than one task simultaneously. Threads are very useful in such situations.

Q3. Define the main thread briefly.

Ans. During the execution of a program, the `JVM` creates a user thread automatically to execute the `main()` method, which is called the `main thread`. If any other user `thread` is not created in the program, it terminates once the `main()` method finishes its execution.

Q4. How are threads created?

Ans. The following two ways are used to create threads in a Java program:

- By extending the java.lang.Thread class
- By implementing the `java.lang.Runnable` interface

Q5. List the important facts that you must remember while creating threads by extending the Thread class.

Ans. The important facts to keep in mind while creating threads by extending the java.lang.Thread class are as follows:

- The class that extends the `Thread` class overrides the `run()` method in which the code executed by the `thread` is defined.
- The `constructor` of the `Thread` class may be called explicitly in the `constructor` of the class extending it by using the `super` keyword to initialize the `thread`.
- The class extending the `Thread` class invokes the `start()` method, which is inherited from the `Thread` class to make the `thread` eligible for running.

Q6. List the events that take place when a thread is invoked.

Ans. The following events take place when the start() method of a `thread` object is invoked:

- A new thread of execution starts with a new call stack.
- The state of the thread changes from the new state to the runnable state.
- The run() method is executed when the thread gets its turn.

Q7. What is a thread scheduler?

Ans. The component of the `JVM` that decides the order in which threads are executed is called the `thread scheduler`. Even though you have multiple threads in your program, the system that has a single processor executes only a single thread at a particular point of time. This means that even if there are multiple active threads, then the resources are allocated only to the thread that is executing. The thread `scheduler` selects a thread for execution from the `runnable` pool and sends back the executing thread to the `runnable` pool depending on the priority of the threads and the code inside the `run()` method of the thread.

Q8. What is a deadlock?

Ans. When multiple threads are used in a program, they share the resources. A particular resource is allocated to a thread that acquires the Object lock for the resource. `Object locks` are variables that are acquired by a thread from the runnable pool. A deadlock is a situation where two threads wait for the Object locks acquired by each other. The Object locks are not released by the respective threads before completing their execution. This situation, where threads are waiting for the Object lock forever, leads to a deadlock.

Q9. Define locks. How do they implement mutual exclusion?

Ans. The synchronization of code in a `Java` program is done with the help of `locks`. Every object in the Java programming language has a default `object lock`, which can be used for synchronization. Threads gain access to a shared object by first acquiring the `lock` associated with them. After a thread acquires the lock associated with an object, the other threads are prevented from acquiring the lock associated with the same object at that point of time. This guarantees that only one thread modifies the state of an object at a time. The Object lock, also known as

monitor, allows only one thread to use the shared resources at a time. In this way, the lock implements mutual exclusion.

Q10. List the rules that must followed while synchronizing shared resources.

Ans. The following rules must be followed during the synchronization of shared resources:

- A thread must acquire the object lock associated with a shared object before it uses the shared object. The runtime system ensures that if a thread has the lock of the shared object, it will not allow another thread to access the shared object. If a thread needs to access the shared resource at the same time, it is blocked and has to wait until the resource becomes available.
- When a thread completes its execution and finishes modifying the shared resource, the runtime system again ensures that the lock of the shared resource has been relinquished by the thread that was using the resource. The resource is then assigned to the thread waiting for it.
- Methods or blocks can be synchronized, but not variable or classes.
- Only one lock is associated with each object/shared resource.

Q11. List the packages in the Java API.

Ans. The packages available in the Java API are:

- java.lang
- java.io
- java.util
- java.net
- java.awt
- java.applet

Q12. Define the constructors of the Random class.

Ans. The following are the classes of the Random class:

- **public Random()**: Constructs a new random number generator.
- **public Random(long seed)**: Constructs a new random number generator by using a specified seed.

Debugging Exercises

Q1. Find the error in the following program:

```
class ThreadDemo implements Runnable
{
 public void run(Thread thrd)
 {
     for(int i=0;i<5;i++)
     {
         System.out.println("Hello World");
     }
 }
}
public class MyThread
{
```

```
public static void main(String ar[])
{
    ThreadDemo thrd=new ThreadDemo();
    Thread newThread=new Thread(thrd);
    newThread.start();
}
}
```

Ans. The preceding program generates a compilation error because the run() method of the Runnable interface has not been implemented. Although an overloaded version of the run() method is implemented, it is not recognized by a runnable thread. The compiler therefore generates an error stating that the run() method is not overridden in the ThreadDemo class.

Q2. Will the following program compile? If not, why?

```
class Test implements Runnable
{
 public void run()
 {
     for(int i=1; i<=5;i++)
     {
         System.out.println("The current value= "+i);
         Thread.sleep(1000);
     }
 }
}
```

Ans. The program generates a compilation error because the sleep() method generates the InterruptedException exception even though it is not invoked from within the try/catch block.

Q3. What will be the output of the following program?

```
public class Test implements Runnable{
 demo d;
 public static void main(String ar[])
 {
     new Test().disp();
 }
 void disp(){
     d=new demo();
     new Thread(new Test()).start();
     new Thread(new Test()).start();

 }
 public void run(){
     d.fun(Thread.currentThread().getId());
 }
}
class demo
{
 static long f=0;
 synchronized void fun(long id){
     for(int i=1;i<3;i++){
         System.out.println(id+" ");
         Thread.yield();
```

```
    }
  }
}
```

Ans. When you compile and execute the preceding program, it will compile successfully but throw a runtime exception because the d reference variable is not static here. When the run() method is invoked, it is with a new instance of the Test class, and the d reference variable of the demo class has not been assigned to an object.

Q4. What will be the output of the following program?

```
public class Test implements Runnable
{
 public static void main(String ar[])
 {
     Thread th= new Thread(new Test());
     th.start();
     System.out.print("A");

     try{
         th.join();
         System.out.print("B");
     }
     catch(Exception e){}
 }
 public void run()
 {
     System.out.print("C");
     System.out.print("D");
  }

}
```

Ans. When you compile and execute the preceding code, "ACDB" is displayed as an output.

Q5. Find the error in the following program:

```
public class Test
{
 static Thread x,y;
 public static void main(String ar[])
 {
     y=new Thread()
     {
         public void run()
         {
             try{
             x.wait(); }
             catch(Exception e) {
                 System.out.print("3"+" ");
             }
             System.out.print("4"+" ");
         }
     };
     x=new Thread(){
         public void run(){
             try{
```

```
                y.sleep(2000);}
                catch(Exception e)
                {
                    System.out.print("1"+" ");
                }
                System.out.print("2" + " ");
            }
        };
        x.start();
        y.start();
    }
}
```

Ans. The program does not generate any error. It compiles successfully and displays "3 4 2" as an output.

Q6. What will be the output of the following program?

```
public class Test
{
 public static void main(String ar[])
 {
     TestDemo t1=new TestDemo()
     {
         public void run()
         {
             System.out.println("Inside Run 3");
         }
     };
     Thread t=new Thread(t1);
     t.start();
 }
}
class TestDemo  implements Runnable
{
 TestDemo() {
 System.out.print("Inside Constructor"+" ");
 }
 public void run()
 {
     System.out.print(" Inside Run 1");
 }
 public void run(String st) {
     System.out.print("Inside Run 2");
 }
}
```

Ans. When you compile and execute the preceding program, it displays "Inside Constructor Inside Run 3" as an output.

Q7. What will be the output of the following program?

```
class Test extends Thread {
 int a;
 public Test(int a) {
     this.a = a;
 }
 public void run() {
```

```
        System.out.println(a);
    }
    public static void main(String args[])
    {
        new Test(10);  //1
    }
}
```

Ans. The preceding program compiles successfully and terminates without displaying any output on execution.

Q8. What will be the output of the following program?

```
class Test implements Runnable
{
public void run()
{
 try
     {
     for (int i = 1; i < 2; i++) .
         {
         Thread.currentThread().sleep(2000);
         System.out.println("Sleep method is testing...");
     }
 } catch (InterruptedException e) {
     System.out.println("Exception is handled in run() method");
 }
}

public static void main(String args[])
{
 Test obj = new Test();
 Thread t1 = new Thread(obj);
 t1.start();
 }
}
```

Ans. When you compile and execute the preceding program, it will generate "Sleep method is testing..." as the output.

Q9. What will be the output of the following program?

```
class Test extends Thread
{
 public Test()
 {
     setPriority(5);
 }
 public void run()
 {
     System.out.println(" Executing Thread");
 }

 public static void main(String args[])
 {
     Test t1 = new Test();
     Test t2 = new Test();
      t1.start();
```

```
        t2.start();
    }
}
```

Ans. The preceding program compiles and executes successfully displaying "Executing Thread" two times as an output.

Q10. What will be the output of the following program?

```
public class Test
{
 public static synchronized void main(String ar[])
 {
     Thread td=new Thread();
     td.start();
     System.out.println("1");
     try{
         td.wait(2000);
         System.out.println("2");}
         catch(InterruptedException e){}
 }
}
```

Ans. When you compile and execute the preceding program, it displays the following output:

```
1
Exception in thread "main" java.lang.IllegalMonitorStateException
 at java.lang.Object.wait(Native Method)
 at Test.main(Test.java:9)
```

The IllegalMonitorStateException is thrown as program does not acquire object lock for the td object. The td object is not synchronized.

Q11. What will be the output of the following program?

```
public class Test extends Thread {
 public void run() {
 }
 public static void main(String args[]) {
     System.out.println("Starting main() method");
     new Test().start();
     System.out.println("Ending main() method");
 }
}
```

Ans. When you compile and execute the preceding program, it displays the following output:

```
Starting main() method
Ending main() method
```

Q12. What will be the output of the following program?

```
class Test extends Thread
{
 String str = "InitialString";
```

```
    public Test(String str) {
        this.str = str;
    }
    public void run() {
        System.out.println(str);
    }
    public static void main(String args[]) {
        new Test("Thread1").run();
        new Test("Thread2").run();
        System.out.println("Thread end");
    }
}
```

Ans. The preceding program compiles and executes successfully and displays the following output:

```
Thread1
Thread2
Thread end
```

Q13. Find the error in the following program:

```
class MyThread extends Thread
{
 public static void main(String args[])
 {
     new MyThread().start();
     new MyThread().start();
 }
 public void run()
 {
     Test t1 = new Test("123");
     Test t2 = new Test("456");
 }
}
class Test {
 private String str;
 public synchronized Test(String s) {
     System.out.print(str);
 }
}
```

Ans. The preceding program generates a compilation error because the constructor of Test class is defined with the synchronized modifier and we cannot use the synchronized keyword with any constructor. However, we can use the synchronize block within the body of a constructor.