Unit III

Packages and Interfaces

Packages are groups of related classes. Packages help us to organize code and provide another layer of encapsulation. An **interface** defines a set of methods that will be implemented by a class. Interface does not implement any method by itself. Packages and interfaces give the users greater control over the organization of programs.

Packages

In programming, it is helpful to group related pieces of program together. In Java, this is done by using a package. Package serves two purposes:

1. It provides a mechanism by which related pieces of a program can be organized as a unit. Classes defined within a package must be accessed through their package name.
2. Package participates in Java's access control mechanism. Classes defined within a package can be made private to that package and not accessible by code outside the package. Thus package provides a means by which classes can be encapsulated.

In general, when a class is named, we are allocating a name from namespace. A namespace defines a declarative region. In Java, no two classes can use the same name from the same namespace. Thus, within a given namespace, each class name must be unique. Since package usually contains related classes, Java defines special access rights to the code within a package. This enables the programmer to create self-contained group of related classes that keep their operation private.

Defining a Package

All classes in Java belong to some package. When no package statement is specified, the default package is used. A package can be created by adding a **package** command at the top of the Java source file. The classes declared within that file will belong to the specified package. Since a package specifies a namespace, the names of the classes inside it become the part of the namespace.

The general form of package statement is

```
package pkg;
```

where pkg is the name of the package.

Example: package myPack;

Accessing a Package

It is to be noted that a Java system package can be accessed either using a fully qualified class name for using a shortcut approach through the import statement. We use import statement when there are many references to a particular package or the package name is too long and unwieldy.

The same approach can be used to access the user-defined packages as well. The ***import*** statement can be used to search a list of packages for a particular class. The general form of import statement for searching a class is :

```
import package1 [.package2] [.package3].classname;
```

Here *package1* is the name of the top level package, *package2* is the name of the package that is inside the *pacakge1*, and so on. We can have any number of packages in a package

hierarchy, Finally, the explicit classname is specified. Note that the statement must end with a semicolon and the import statement should appear before any class definitions in a source file. Multiple import statements are allowed.

Following is an example of importing a particular class :

```
import firstPackage.secondPackage.MyClass;
```

After using this statement, all the members of the class MyClass can be directly accessed using the class name or its objects directly without using the package name.

We can also use another approach as follows

```
import packagename.*;
```

Here, *packagename* may denote a single package or a hierarchy of packages. The * indicate that the compiler should search this entire package hierarchy when it encounters a class name. This implies that we can access all classes contained in the above package directly.

Adding a class to a package.

It is simple to add a classs to an existing package. For example consider

```
package p1;
     public class ClassA
     {
          …..
     }
```

The above package **p1** contains one class **ClassA**. Suppose we want to add another class **ClassB** to this package. This can be done as follows :

1. Define the class and make it public.
2. Place the package statement p1 before the class defintion as follows

```
package p1;
public class ClassB
               {
                    ……
               }
```

Packages and Member Access

The visibility of an element is determined by its access specification—private, public, protected or by default. If a member has no explicit access modifier, then it is visible within its package and not outside the package. The members which are declared s public are visible everywhere including different classes of different package. There is no restriction on their use or access. A private member is available only to the other members of its class. A member specified as protected is accessible within its package and to all subclasses including subclasses in another package.

Class Member Access

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

Built-in Packages

Java defines a large number of standard classes that are available to all programs. This class library is referred to as the Java API (Application Programming Interface) The Java. API is stored in packages. At the top of the package hierarchy is **java.**

Subpackage	**Description**
java.lang	Langage support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	Language utility classes as vectors, hash tables, random numbers, date, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets

Interfaces

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. An interface doesn't provide any implementation. To implement interface, classes must provide bodies for the methods described by the interface. Two classes may implement the same interface in different ways but each class supports the same set of methods. By providing interface keyword, Java allows the programmers to utilize "One interface, multiple methods" aspect of polymorphism.

This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **public**, **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Example:
```
public interface Series {
    int getNext();
   void reset();
   void  setStart(int x);  }
```

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, we must include the **implements** clause in a class definition and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface*...]] {
// class-body
}

```
// Implementing an interface
public interface Series {
        int getNext();
        void reset();
        void  setStart(int x);  }

class ByTow implements Series {
 int start, val;
ByTwo() {
start=0; val=0; }
public int getNext() {
 val+=2;
 return val; }
public void reset() {
 val=start; } public void setStart(int x) {
start=x;
val=x;  } }

class SDemo {
public static void main(String args[]) {
ByTwo ob = new ByTwo();
for(int i=0; i<5; i++)
  System.out.println("Next value is "+ob.getNext());
System.out.println("\n Resetting");
ob.reset();
for(int i=0; i<5; i++)
   System.out.println("Next value is "+ob.getNext());
System.out.println("\n Starting at 100");
ob.setStart(100);
for(int i=0; i<5; i++)
  System.out.println("Next value is "+ob.getNext()); } }
```

Output:

```
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Resetting
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Starting at 100
Next value is 102
Next value is 104
Next value is 106
Next value is 108
Next value is 110
```

Using Interface References

We can create interface reference variables. Such a variable can refer to an object that implements its interface. When a method on an object is called through an interface reference, the version of the version of the method implemented by the object is executed.

```
// Demonstrate interface references
public interface Series {
        int getNext();
        void reset();
        void  setStart(int x);  }
class ByTow implements Series {
 int start, val;

ByTwo() {
start=0; val=0; }

public int getNext() {
 val+=2;
 return val; }
public void reset() {
 val=start; }
```

```
public void setStart(int x) {
start=x;
val=x;  } }
class SDemo2 {
public static void main(String args[]) {
ByTwo two = new ByTwo();
Series ob;
Ob=two;
for(int i=0; i<5; i++)
  System.out.println("Next value is "+ob.getNext());
  } }
```

Extending an Interface

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2()."); }
public void meth3() {
System.out.println("Implement meth3()."); } }
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3(); } }
```

Exception Handling

Exception Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. All exception classes are derived from a class called **Throwable**. There are two direct subclasses of Throwable class: **Exception** and **Error**. Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Program statements that require monitoring are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. The code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, the keyword **throw** can be used. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

Using try and catch

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

Here, ExceptionType is the type of exception that has occurred. When an exception is thrown, it is caught by the corresponding catch statement which then processes the exception. When no exception is thrown, try block ends normally and catch statements are bypassed. Thus catch statements are executed only if an exception is thrown.

A Simple Exception Example

```
// Demonstrate Exception Handling
class EDemo1 {
public static void main(String args[]) {
int nums[] = new int[4];
try {
System.out.println("Before exception is generated");
nums[7]=10; // code that generates exception
System.out.println("This won't be displayed"); }
 catch (ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out of bounds !! "); }
System.out.println("After the catch statement"); } }
```

Output:

```
Before exception is generated
Array index out of bounds !!
After the catch statement
```

The Consequences of an Uncaught Exception

If an exception is caught, it prevents abnormal program termination. If a program doesn't catch any exception, it will be caught by the JVM which terminates the program and displays stack trace and an error message.

```
class NotHandled {
public static void main(String args[]) {
int nums[] = new int[4];
System.out.println("Before exception is
generated");
nums[7]=10; // code that generates exception } }
```

Output:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException : 7 at NotHandled.main(NotHandled.
java : 9

Use of Exception Handling

The main benefit of using exception handling is that, it enables the program to respond to an error and then continue running.

```
// To handle exceptions gracefully
class EDemo2 {
public static void main(String args[]) {
int nr[] {4, 8, 16, 32, 64, 128};
int dr[] {2, 0, 4, 4, 0, 8};
for(int i=0; i<nr.length; i++) {
try
{
System.out.println( nr[i] +"/" + dr[i] +"=" +nr[i]/dr[i]); }
 catch (ArithmeticException  e) {
System.out.println("Can't Divide by zero "); }
} } }
```

Output:
4/2 = 2
Can't divide by zero
16/4=4
32/4=8
Can't divide by zero
128/8= 16

Using Multiple catch Statements

```
// Multiple catch statements
class EDemo3 {
public static void main(String args[]) {
int nr[] {4, 8, 16, 32, 64, 128, 265, 512};
int dr[] {2, 0, 4, 4, 0, 8};
for(int i=0; i<nr.length; i++) {
try
{
System.out.println( nr[i] +"/" + dr[i] +"=" +nr[i]/dr[i]); }
 catch (ArithmeticException  e) {
System.out.println("Can't Divide by zero "); }
catch (ArrayIndexOutOfBoundsException  e) {
System.out.println("No Matching Element "); }
} } }
```

Output:
4/2 = 2
Can't divide by zero
16/4=4
32/4=8
Can't divide by zero
128/8= 16
No Matching Element
No Matching Element

Catching Subclass Exceptions

A catch clause for a superclass will also match any of its subclasses. Since the superclass of all exceptions is Throwable, to catch all possible exception, we can use Throwable. If subclass exception has to be caught, it must be put first in the catch sequence.

```
class EDemo4 {
public static void main(String args[]) {
int nr[] {4, 8, 16, 32, 64, 128, 265, 512};
int dr[] {2, 0, 4, 4, 0, 8};
for(int i=0; i<nr.length; i++) {
try
{
```

Output:
4/2 = 2
Some Exception Occurred
16/4=4 32/4=8
Some Exception Occurred
128/8= 16
No Matching Element
No Matching Element

```
System.out.println( nr[i] +"/" + dr[i] +"=" +nr[i]/dr[i]); }
 catch (ArrayIndexOutOfBoundsException  e) {
System.out.println("No Matching Element "); }
catch (Throwable e) {
System.out.println("Some Exception occurred "); }
} } }
```

Nesting of try Blocks

One try block can be nested within another. An exception generated by the inner try block is not caught by the catch statement associated with that try.

```
class EDemo5 {
public static void main(String args[]) {
int nr[] {4, 8, 16, 32, 64, 128, 265, 512};
int dr[] {2, 0, 4, 4, 0, 8};
try
{
for(int i=0; i<nr.length; i++) {
try {
 System.out.println( nr[i] +"/" + dr[i] +"="
+nr[i]/dr[i]);
}
 catch (ArithmeticException  e) {
System.out.println("Can't divide by zero "); } } }
catch (ArrayIndexOutOfBounds e) {
System.out.println("No matching element ");
System.out.println("Fatal Error \n Program terminated ");}
} }
```

Output:
4/2 = 2
Can't divide by zero
16/4=4
32/4=8
Can't divide by zero
128/8= 16
No Matching Element
Fatal Error
Program terminated

In the example given above, if division by zero exception is occurred, it is handled by the inner catch block and the program execution continues. However, the array boundary error is caught by the outer catch, which causes the program to terminate.

Throwing an Exception

It is possible to throw an exception manually using the **throw** statement.

// manually throw and exception

```
calss ThrowDemo {
public static void main(String args[]) {
try{
System.out.println("Before throw. ");
throw new ArithmeticException(); }
catch (ArithmeticException  e) {
System.out.println("Exception caught "); }
System.out.println("After try /catch block "); }
```

Output:
Before throw.
Exception caught
After try/catch block

Rethrowing an Exception

An exception caught by catch statement can be rethrown so that it can be caught by an outer catch. The reason for rethrowing is to allow multiple handlers.

The Throwable Class

All exceptions are subclasses of Throwable. Throwable supports several methods. The commonly used methods defined by Throwable class are listed below:

Method	Description
Throwable fillinStackTrace()	Returns a Throwable object that contains complete stack trace
String getLocalizedMessage()	Returns a localized description of the exception
void printStackTrace()	Displays the stack trace
void printStackTrace(PrintStream s)	Sends the stack trace to specified stream
void printStackTrace(PrintWriter s)	Sends the stack trace to specified stream
String toString()	Returns the string object containing a complete description of the exception. This method is called by println() when outputting Throwable object.

Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's **throws** list. In the language of Java, these are called ***unchecked exceptions*** because the compiler does not check to see if a method handles or throws these exceptions. Exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called ***checked exceptions***.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException.	Type not found
UnsupportedOperationException	An unsupported operation was encountered
The unchecked exceptions defined in **java.lang**	

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable
IllegalAccessException.	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
The checked exceptions defined in **java.lang**	

Using Finally

To specify a block of code when a try/catch block is exited, **finally** statement can be used. When finally block is included, this is guaranteed to execute whether or not an exception is thrown. Whose general form is

```
try {
// block of code to monitor for errors  }
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1  }
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2  }
// ...
finally {
// block of code to be executed after try block ends  }
```

The **finally** block will be executed whenever the execution leaves a try/catch block.

// Use of finally

```
class UseFinally {
public static void main(Sting args[]) {
int a[]={5,10};
try{
int x=a[2]/b—a[1]; }
catch(ArithmeticException e) {
System.out.println("Division by zero"); }
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array Index Error"); }
catch(ArrayStoreException e) {
System.out.println("Wrong Data type"); }
finally{
int y=a[1]/a[0];
System.out.println("Y = "+y);
```

Output:
Array Index Error
Y = 2

Creating Exception Subclasses (Creating User-Defined Exceptions)

Although Java's built-in exceptions handle most common errors, we can create our own exception types to handle situations specific to our applications. This is quite easy to do: just

define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). These subclasses don't need to actually implement anything—it is their existence in the type system that allows us to use them as exceptions. The **Exception** class does not define any methods of its own. It inherits those methods provided by **Throwable**.

//Demonstrate user-defined exception handling

```
import java.lang.Exception
class MyException extends Exception
{
  MyException(String mag)
      super(msg); } }
class UserException {
public static void main(String args[])
{
  int x=5, y=1000;
try{
 float z = float(x)/float(y);
if(z<0.01)
 {
      throw new MyException("Number is too small"); }
  }
catch(MyException e)
{
      System.out.println("Caught My Exception");
      System.out.println(e.getMessage()); }
finally{
      System.out.println("I am always here"; }
} }
```

Output:
Caught My Exception
Number is too small
I am always here

Multithreaded Programming

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a ***thread***, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Multithreading Fundamentals

There are two distinct types of multitasking: ***process-based*** and ***thread-based***. A process is a program that is executing. A process-based multi-tasking allows the computer to run two or more programs concurrently. In thread-based multi tasking environment, a single program can perform two or more tasks at the same time. The advantage of multithreading is that it makes one to write efficient programs which utilize the idle time that is present in most of the programs. I/O devices are much slower than the CPU. Sometimes, a program spends majority of its execution time waiting to send or receive the information to or from a device. By using multithreading, the program can execute another task.

Threads exist in several states. A thread can be *running*. It can be ready to run as soon as it gets CPU time. A running thread can be *suspended*, which temporarily suspends its activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be *terminated*, which halts its execution immediately. Once terminated, a thread cannot be resumed.

The Thread Class and Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread. To create a new thread, the program will either extend **Thread** or implement the **Runnable** interface. The **Thread** class defines several methods that help manage threads. Few of them are listed below:

Method	**Meaning**
final String getName ()	Obtain a thread's name.
final int getPriority ()	Obtain a thread's priority.
final isAlive()	Determine if a thread is still running.
final void join()	Wait for a thread to terminate.
void run()	Entry point for the thread.
static void sleep(long milliseconds)	Suspend a thread for a period of time.
void start()	Start a thread by calling its run method.

All processes have at least one thread of execution, which is usually called the main thread, because it is the one that is executed when the program begins. From the main thread other threads can be created.

Creating a Thread

There are two methods in Java by which we can create threads.

1. By implementing Runnable interface.
2. By extending Thread class.

The Runnable interface defines only one method called run()

// Demonstrate creation of a Thread

```
class X implements Runnable {
public void run() {
for(int i=0; i<=5; i++)
System.out.println("I from thread X = "+ i); } }
class ThreadDemo {
public static void main(String args[]) {
X rn = new X();
Thread t1=new Thread(rn);
t1.start();
System.out.println("End of main"); } }
```

Output
I from Thread X = 0
I from Thread X = 1
I from Thread X = 2
I from Thread X = 3
I from Thread X = 4
I from Thread X = 5
End of main

Creating Multiple Threads

The above program creates a single thread. Multiple threads can be created in same way. The following program illustrates creation of multiple threads.

```
class Square implements Runnable {
public void run() {
for(int i=1; i<=5; i++)

System.out.println("Square of "+ i + " = "+ i*i); }
 }
class Cube implements Runnable {
public void run() {
for(int i=1; i<=5; i++)
System.out.println("Cube of "+ i + " = "+ i*i*i); }
 }
class ThreadDemo1 {
public static void main(String args[]) {
Square  s = new Square();
Cube c = new Cube();
Thread t1=new Thread(s);
Thread t2=new Thread(c);
t1.start();
t2.start();
System.out.println("End of main"); } }
```

Output
Square of 1 = 1
Square of 2 = 4
Square of 3 = 9
Square of 4 = 16
Square of 5 = 25
Cube of 1 =1
Cube of 2 =8
Cube of 3 =27
Cube of 4 =64
Cube of 5 =125
End of main

Determining When a Thread Ends

Thread provides two means by which we can determine whether the thread has ended.

1. The isAlive() method which waits for the child thread to terminate.
2. The join() method which waits until the thread on which it is called terminates. The calling thread is waiting until the specified thread joins it.

Thread Priority

Each thread is associated with a priority. The priority of a thread determines how much CPU time a thread receives relative to other active threads. Low-priority threads receive a little and high-priority threads receive a lot of CPU time. When high priority thread waits for a resource, low-priority thread may run. When high priority thread gains access to the resource, it may preempt the low-priority thread and resume execution.

When a child thread is started, its priority setting is equal to that of its parent thread. The priority of a thread can be changed by calling setPriority(). This method has general form

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range MIN_PRIORITY and MAX_PRIORITY with values 1 and 10 respectively. The defaultpriority value is NORM_PRIORITY which is 5.
The current priority of a thread can be obtained by using getPriority() whose general form is

```
final int getPriority()
```

```
//Demonstrate Thread Priority
class A extends Thread {
public void run() {
for(int i=0; i<=4;i++)
   System.out.println("From Thread A i = "+i);
System.out.println("Exit from A"); } }
class B extends Thread {
public void run() {
for(int j=0; j<=4;j++)
   System.out.println("From Thread B j = "+j);
System.out.println("Exit from B"); } }
class C extends Thread {
public void run() {
for(int k=0; j<=4;k++)
   System.out.println("From Thread C k = "+k);
System.out.println("Exit from C"); } }
class ThreadPriority {
public static void main(String args[]) {
A threadA = new A();
B threadB = new B();
C threadC = new C();
threadC.setPriority(Thread.MAX_PRIORITY);
threadB.setPriority(threadA.getPriority()+1);
threadA.setPriority(Thread.MIN_PRIORITY);
System.out.println("Thread A Started");
threadA.start();
System.out.println("Thread B Started");
threadB.start();
System.out.println("Thread C Started");
threadC.start();  } }
```

In the above example, thread is assigned with priority value 10, thread with 6 and threadA with 1 respectively.

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called ***synchronization***. The most common reason for synchronization is when two or more threads need to access to a shared resource that can be used by only one thread at a time. For example, when a thread is writing to a file, the second thread must be prevented from doing so at the same time. Java provides unique, language-level support for it.
Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor.

All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. There are two ways to synchronize a code.

1. Using Synchronized Methods
2. Using Synchronized Statement

Synchronized Methods

We can make a method synchronized by prefixing the keyword synchronized. When a method is called, the calling thread enters the object's monitor, which locks the object. While locked, no other object can enter the method. When the thread enters from the method, the monitor unlocks the object, allowing it to be used by another thread.

```
//Use of Synchronize to control the access
import java.io.*;
class SumArray{
  private int sum;
  synchronized int SumArray(int nums[]) {
    sum=0;
    for(int i=0; i<nums.length; i++) {
      sum+=nums[i];
      System.out.println("Running total for"
Thread.currentThread().getName() + "is " + sum);
      try{
        Thread.sleep(100)   }
      catch(InterruptedException e)
      { System.out.println("Thread Interrupted");   } }
  return (sum); } }
class MyThread implements Runnable
{ Thread thrd;
  static SumArray sa = new SumArray();
  int a[];
  int answer;
  MyThread(String Name, int nums[]) {
    thrd=new Thread(this, Name);
    a=nums;
    thrd.start(); }
  public void run(){
    int sum;
    System.out.println(thrd.getName() + "Starting");
    answer=sa.SumArray(a);
    System.out.println("Sum for "+ thrd.getName() + answer);
    System.out.println(thrd.getName() + "Terminating");  } }
class Sync {
  public static void main(String args[]) {
    int a[]={1,2,3,4,5,6,7,8};
    int b[]={1,3,5,7,9};
    MyThread t1 = new MyThread3("Child #1", a);
    MyThread t2 = new MyThread3("Child #2", b);
    try {
      t1.thrd.join();
      t2.thrd.join(); }
    catch(InterruptedException e) {
      System.out.println("Main Thread Interrupted !!"); } }}
```

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

    // Synchronized method to increment the
counter
    public synchronized void increment() {
        count++;
        System.out.println("Counter value: " +
count);
    }
}

class CounterThread extends Thread {
    private Counter counter;

    public CounterThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            counter.increment();
            try {
                Thread.sleep(100); // Just to simulate
some processing time
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        // Create multiple threads to increment the
counter
        CounterThread thread1 = new
CounterThread(counter);
        CounterThread thread2 = new
CounterThread(counter);

        // Start the threads
        thread1.start();
        thread2.start();
    }
}
```

The Synchronized Statement

While creating **synchronized** methods within classes that is an easy and effective means of achieving synchronization, it will not work in all cases. For example, if we want to synchronize access to objects of a class that was not designed for multithreaded access that is, the class does not use **synchronized** methods. Further, this class was not created by us , but by a third party and we do not have access to the source code. Thus, we can't add **synchronized** to the appropriate methods within the class. The solution to this problem is quite easy: We simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

synchronized(*object*) {

// statements to be synchronized }

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

Thread Communication Using wait(), notify() and NotifyAll()

When a thread T is executing inside a synchronized method and needs access to a resource R which is temporaritly unavailable. Then T enters into the polling loop that waits for R, T ties up the object preventing other threads' access to it. This solution is less optimal. A better solution is gien by an elegant inter-process communication mechanism via the **wait()**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

• **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.

• **notify( )** wakes up a thread that called **wait( )** on the same object.

• **notifyAll( )** wakes up all the threads that called **wait( )** with the highest priority thread gaining access to object.

These methods are declared within **Object**, as shown here:

final void notify( )

final void notifyAll( )

Additional forms of **wait( )** exist that allow us to specify a period of time to wait.

Although **wait( )** normally waits until **notify( )** or **notifyAll( )** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify( )** or **notifyAll( )** having been called. In essence, the thread resumes for no apparent reason.

Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter. The methods used achieve this are suspend() resume() anhd stop(). They have the following form:

final void resume()

final void suspend()

final void stop()