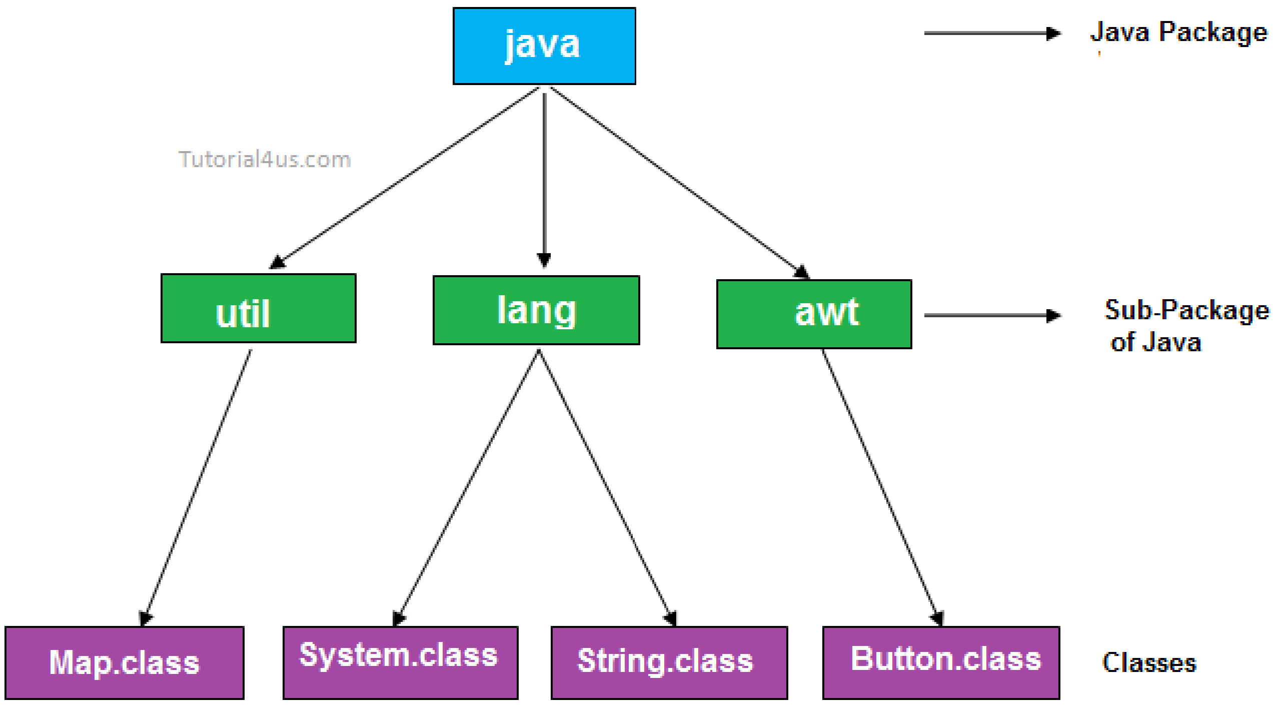
CHAPTER-5
PACKAGES AND MULTITHREADING



Packages

Introduction to packages:

1) The package contains group of related classes and interfaces.
2) The package is an encapsulation mechanism it is binding the related classes and interfaces.
3) We can declare a package with the help of package keyword.
4) Package is nothing but physical directory structure and it is providing clear-cut separation between the project modules.
5) Whenever we are dividing the project into the packages(modules) the sharability of the project will be increased.

Syntax:-

Package package_name;
Ex:- package com.dss;

The packages are divided into two types

1) Predefined packages
2) User defined packages

Predefined packages:-

The java predefined packages are introduced by sun peoples these packagescontains predefined classes and interfaces.

Ex:- java.lang
Java.io
Java.awt
Java.util
Java.net.........................etc

Java.lang:-

The most commonly required classes and interfaces to write a sample program is encapsulated into a separate package is called java.lang package.

` Ex:- String(class)
StringBuffer(class)
Object(class)
Runnable(interface)
Cloneable(nterface)

Note:-
the default package in the java programming is java.lang if we are importing or not importing by default this package is available for our programs.

Java.io package:-

The classes which are used to perform the input output operations that are present in the java.io packages.

Ex:- FileInputStream(class)

FileOutputStream(class)
FileWriter(class)
FileReader(class)

Java.net package:-

The classes which are required for connection establishment in the network that classes are present in the java.net package.

Ex:- Socket
ServerSocket
InetAddress
URL

Java.awt package:-

The classes which are used to prepare graphical user interface those classes are present in the java.awt package.

Ex: Button(class)
Checkbox(class)
Choice(Class)
List(class)

User defined packages:-

1) The packages which are declared by the user are called user defined packages.
2) In the single source file it is possible to take the only one package. If we are trying to take two packages at that situation the compiler raise a compilation error.
3) In the source file it is possible to take single package.
4) While taking package name we have to fallow some coding standards.

Whenever we taking package name don't take the names like pack1, pack2, sandhya, sri……… these are not a proper coding formats.

Rules to follow while taking package name:-(not mandatory but we have to follow)

1) The package name is must reflect with your organization name. The name is reverse of the organization domain name.

Domain name:-	**www.dss.com**
Package name:-	**Package com.dss;**

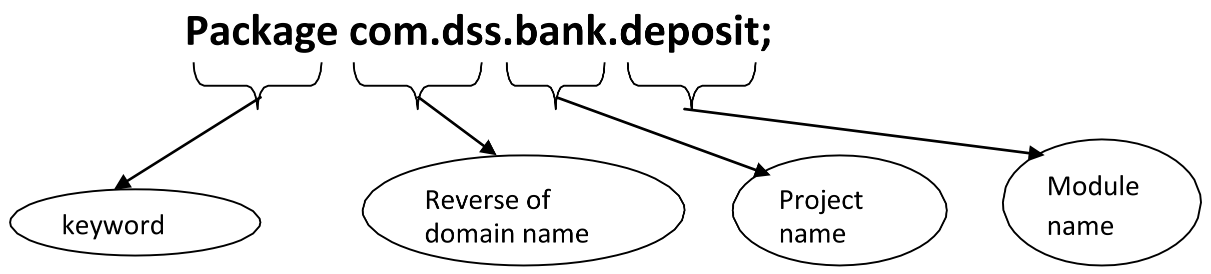
2) Whenever we are working in particular project(Bank) at that moment we have to take the package name is as fallows.

Project name :-	**Bank**
package :-	**Package com.dss.Bank;**

3) The project contains the module (deposit) at that situation our package name should reflect with the module name also.

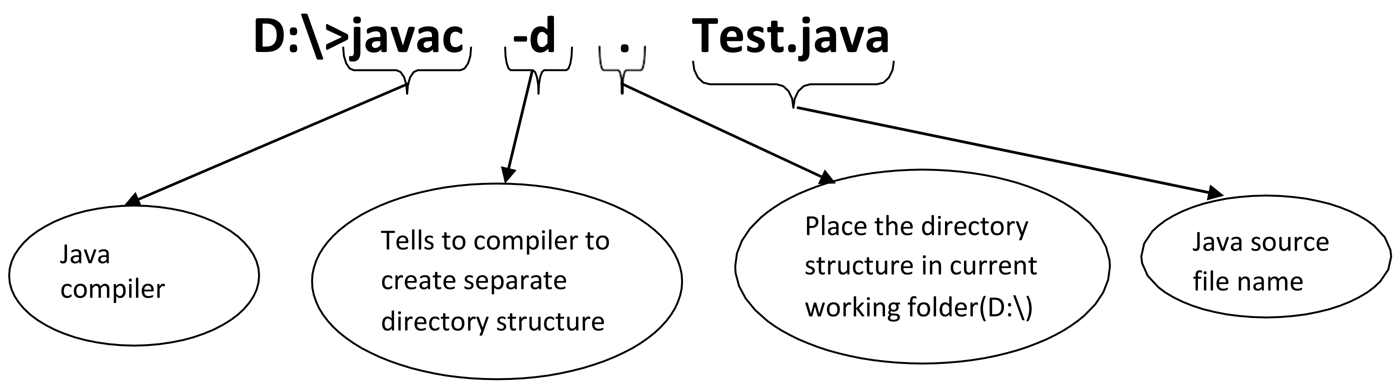
Domain name:-	**www.dss.com**
Project name:-	**Bank**
Module name:-	**deposit**
package name:-	**Package com.dss.bank.deposit;**

For example the source file contains the package structure is like this:-



Note:-

If the source file contains the package statement then we have to compile that program with the help of fallowing statements.


D:\>javac -d . Test.java
Java compiler
Tells to compiler to create separate directory structure
Place the directory structure in current working folder(D:\)
Java source file name

After compilation of the code the folder structure is as shown below.

```
Com
   |
   | -----dss
            |
            | ------bank
                       |
                       | ---deposit
                              |
                              | ------- (number of .classes will be generated )
                              |
```

Note :-

If it a predefined package or user defined package the packages contains number of classes.

Ex 1:-

```
package com.dss.bank.deposit;
class Test
{
	public static void main(String[] args)
	{
		System.out.println("package example program");
	}
}
```

Compilation : javac –d . Test.java

```
Com
   |
   | ----- dss
            |
            | ------ bank
                      |
                      | --- deposit
                             |
                             | ------- Test.class
```

Execution :java com.dss.bank.deposit.Test

Ex:- (compilation error)

```
package com.dss.bank.deposit;
package com.dss.online.corejava;
class Test
{
	public static void main(String[] args)
	{
		System.out.println("package example program");
	}
}
```

Reason:-

Inside the source file it is possible to take the single package not possible to take the multiple packages.

Ex 2:-

```
package com.tcs.OnlineExam.corejava;
class Test
{
	public static void main(String[] args)
	{
		System.out.println("package example program");
	}
}
class A
{
}
class B
{
}
class C
{
}
```

Compilation :- javac –d . Test.java

```
Com
  |
  | ----- tcs
           |
           | ------ OnlineExam
                       |
                       | --- corejava
                              |
                              | ------- Test.class
                              | ------- A.class
                              | ------- B.class
                              | ------- C.class
```

Execution :- java com.tcs.onlineexam.Test

Note:-

The package contains any number of .classes the .class files generation totally depends upon the number of classes present on the source file.

Import session:-

The main purpose of the import session is to make available the java predefined support into our program.

Predefined packages support:-

Ex1:-

```
Import java.lang.String;
```

String is a predefined class to make available predefined string class to the our program we have to use import session.

Ex 2:-

```
Import java.awt.*;
```

To make available all predefined class present in the awt package into our program. That * represent all the classes present in the awt package.

User defined packages support:-

I am taking two user defined packages are

```
1)  Package pack1;
        Class A
        {
        }
        Class B
        {
        }
2)  Package pack2
        Class D
        {
        }
```

Ex 1:-

```
        Import pack1.A;
```

A is a class present in the pack1 to make available that class to the our program we have to use import session.

Ex 2:-

```
        Import pack1.*;
```

By using above statement we are importing all the classes present in the pack1 into our program. Here * represent the all the classes.

Note:-

If it is a predefined package or user defined package whenever we are using that package classes into our program we must make available that package into our program with the help of import statement.

Public:-

- This is the modifier applicable for classes, methods and variables (only for instance and static variables but not for local variables).
- If a class is declared with public modifier then we can access that class from anywhere (within the package and outside of the package).
- If we declare a member(variable) as a public then we can access that member from anywhere but Corresponding class should be visible i.e., before checking member visibility we have to check class visibility.

Ex:-

```
public class Test           // public class can access anywhere
{
        public int a=10; //public variable can access any where
        public void m1()            //public method can access any where
        {
                System.out.println("public method access in any package");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.m1();
                System.out.println(t.a);
        }
}
```

Default:-

- This is the modifier applicable for classes, methods and variables (only for instance and static variables but not for local variables).
- If a class is declared with <default> modifier then we can access that class only within that current package but not from outside of the package.
- Default access also known as a package level access.
- The default modifier in the java language is **default.**

Ex:-

```
class Test
{
        void m1()
        {
                System.out.println("m1-method");
        }
        void m2()
        {
                System.out.println("m2-method");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.m1();
                t.m 2();
        }
}
```

Note :-

in the above program we are not providing any modifier for the methods and classes at that situation the default modifier is available for methods and classes that is default modifier. Hence we can access that methods and class with in the package.

Private:-

- private is a modifier applicable for methods and variables.
- If a member declared as private then we can access that member only from within the current class.
- If a method declare as a private we can access that method only within the class. it is not possible to call even in the child classes also.

```
class Test
{
        private void m1()
        {
                System.out.println("we can access this method only with in this class");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
```

```
                    t.m1();
          }
}
```

Protected :-

- If a member declared as protected then we can access that member with in the current package anywhere but outside package only in child classes.
- But from outside package we can access protected members only by using child reference. If we try to use parent reference we will get compile time error.
- Members can be accesses only from instance area directly i.e., from static area we can't access instance members directly otherwise we will get compile time error.

Ex:-demonstrate the user defined packages and user defined imports.
krishna project source file:-

```
package com.dss;
public class StatesDemo
{
          public void ap()
          {
                    System.out.println("ANDHRA PRADESH");
          }
          public void tl()
          {
                    System.out.println("TELENGANA");
          }
          public void tn()
          {
                    System.out.println("TAMILNADU");
          }
}
```

Tcs project source file:-

```
package com.tcs;
import com.dss.StatesDemo;//or import com.dss.*;
public class StatesInfo
{
          public static void main(String[] args)
          {
                    StatesDemo sd=new StatesDemo();
                    sd.ap();
                    sd.tl();
                    sd.tn();
          }
}
```

Step 1 :- javac -d . StatesDemo.java

Step 2 :- javac -d . StatesInfo.java

Step 3 :- java com.tcs.StatesInfo

Static import:-

1) this concept is introduced in 1.5 version.

2) if we are using the static import it is possible to call static variables and static methods directly to the java programming.

Ex:-without static mport

```
import java.lang.*;
class Test
{
	public static void main(String[] args)
	{
		System.out.println("Hello
World!");
	}

}
```

Ex :- with static import

```
import static java.lang.System.*;
class Test
{
	public static void main(String[] args)
	{
	out.println("ratan world");
	}
}
```

```
Ex:-package com.dss;
public class Test
{
	public static int a=100;
	public static void m1()
	{
	System.out.println("m1 method");
	}
};
```

```
Ex:-
package com.tcs;
import static com.dss.Test.*;
class Test1
{
	public static void main(String[] args)
	{
		System.out.println(a);
		m1();
	}
}
```

Source file Declaration rules:-

The source file contains the fallowing elements

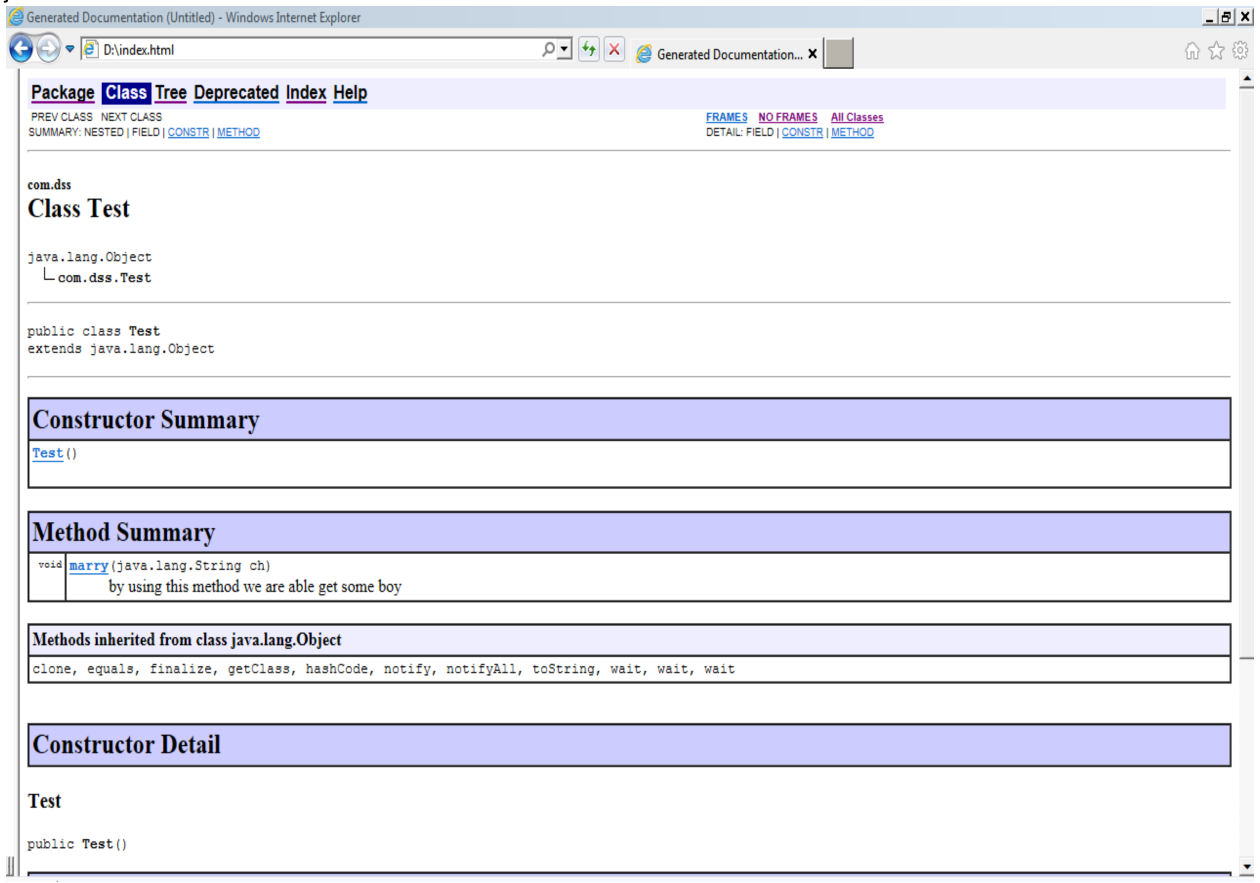
1) Package declaration---→optional ---- →at most one package(0 or 1)--→1st statement
2) Import declaration-----→optional-----→any number of imports ------- →2nd statement
3) Class declaration--------→optional-----→any number of classes-------- →3rd statement
4) Interface declaration---→optional----→any number of interfaces ----→3rd statement
5) Comments declaration-→optional----→any number of comments --- →3rd statement

a. The package must be the first statement of the source file and it is possible to declare at most one package within the source file .
b. The import session must be in between the package and class statement. And it is possible to declare any number of import statements within the source file.
c. The class session is must be after package and import statement and it is possible to declare any number of class within the source file.
    i. It is possible to declare at most one public class.
    ii. It is possible to declare any number of non-public classes.
d. The package and import statements are applicable for all the classes present in the source file.
e. It is possible to declare comments at beginning and ending of any line of declaration it is possible to declare any number of comments within the source file.

Preparation of userdefined API (application programming interface document):-

1. API document nothing but user guide.
2. Whenever we are buying any product the manufacturing people provides one document called user guide. By using userguide we are able to use the product properly.
3. James gosling is developed java product whenever james gosling is deliverd the project that person is providing one user document called API(application programming interface) document it contains the information about hoe to use the product.
4. To prepare userdefined api document for the userdefined projects we must provide the description by using documentation comments that information is visible in API document.
5. If we want to create api document for your source file at that situation your source file must contains all members(classes,methods,variables….) with modifiers.

```
package com.dss;
/*
*parent class used to get parent values
*/
public class Test extends Object
{
        /** by using this method we are able get some boy*/
        public void marry(String ch)
        {
                System.out.println("xxx boy");
        }
}
```



Generated Documentation (Untitled) - Windows Internet Explorer
D:\index.html
Generated Documentation...
Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD
FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

com.dss
Class Test

```
java.lang.Object
  └com.dss.Test
```

```
public class Test
extends java.lang.Object
```

Constructor Summary

Test()

Method Summary

void	marry(java.lang.String ch) by using this method we are able get some boy

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Test

public Test()

Multi Threading

Introduction

1) In the earlier days, the computer's memory is occupied by only one program. After completion of one program, it is possible to execute another program. It is called **uniprogramming**.
2) Whenever one program execution is completed then only second program execution will be started. Such type of execution is called cooperative execution, this execution we are having lot of disadvantages.
    a) Most of the times memory will be wasted.
    b) CPU utilization will be reduced because only program allow executing at a time.
    c) The program queue is developed on the basis cooperative execution

To overcome above problem, a new programming style was introduced and is called as multiprogramming.

Multiprogramming means executing the more than one program at a time.

1) All these programs are controlled by the CPU scheduler.
2) CPU scheduler will allocate a particular time period for each and every program.
3) Executing several programs simultaneously is called multiprogramming.
4) Multiprogramming mainly focuses on the number of programs.
5) In multiprogramming a program can be entered in different states.
    a. Ready state.
    b. Running state.
    c. Waiting state.

Advantages of multiprogramming:-

1. The main advantage of multithreading is to provide simultaneous execution of two or more parts of a application to improve the CPU utilization.
2. CPU utilization will be increased.
3. Execution speed will be increased and response time will be decreased.
4. CPU resources are not wasted.

Thread:-

1. Thread is nothing but separate path of sequential execution.
2. The independent execution technical name is called thread.
3. Whenever different parts of the program executed simultaneously that each and every part is called thread.
4. The thread is light weight process because whenever we are creating thread it is not occupying the separate memory it uses the **same memory**. Whenever the memory is shared means it is not consuming more memory.

Executing more than one thread a time is called multithreading.

Multitasking: Executing several tasks simultaneously is the concept of multitasking. There are two types of multitasking.

1. **Process based multitasking.**
2. **Thread based multitasking.**

Multitasking
Process based multitasking
Thread based Multithreading

Process based multitasking:

Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking.
Example:

- While typing a java program in the editor we can able to listen mp3 audio songs at the same time we can download a file from the net all these tasks are independent of each other and executing simultaneously and hence it is Process based multitasking.
- This type of multitasking is best suitable at "os level".

Thread based multitasking:

Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based multitasking.
And each independent part is called a "Thread".

1. This type of multitasking is best suitable for "programmatic level".
2. When compared with "C++", developing multithreading examples is very easy in java because java provides in built support for multithreading through a rich API (Thread, Runnable, ThreadGroup, ..etc).
3. In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.
4. Whether it is process based or Thread based the main objective of multitasking is to improve performance of the system by reducing response time.

The main important application areas of multithreading are:

1. To implement multimedia graphics.
2. To develop animations.
3. To develop video games etc.
4. To develop web and application servers.

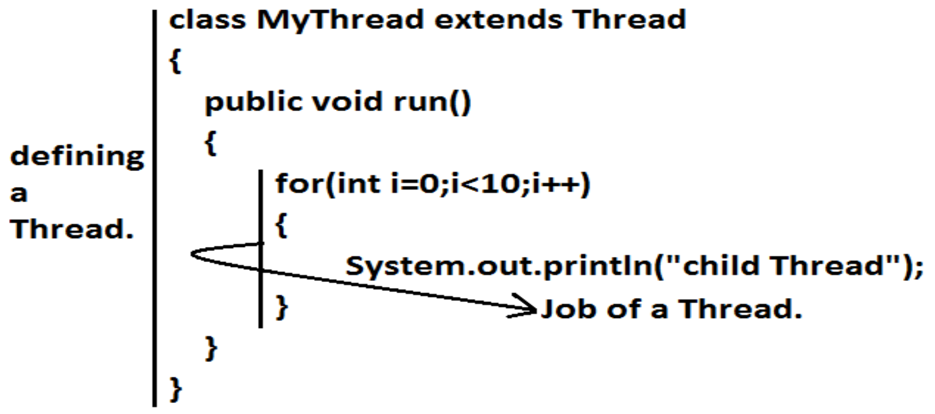
The ways to define instantiate and start a new Thread:

We can define a Thread in the following 2 ways.

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

Defining a Thread by extending "Thread class":

Example:



```
class MyThread extends Thread
{
   public void run()
   {
      for(int i=0;i<10;i++)
      {
            System.out.println("child Thread");
      }
   }
}
```

```
class ThreadDemo
{
	public static void main(String[] args)
	{
		MyThread t=new MyThread();//Instantiation of a Thread
		t.start();//starting of a Thread

		for(int i=0;i<5;i++)
		{
			System.out.println("main thread");
		}
	}
}
```

Case 1: Thread Scheduler:

- If multiple Threads are waiting to execute then which Thread will execute 1st is decided by "Thread Scheduler" which is part of JVM.

- Which algorithm or behavior followed by Thread Scheduler we can't expect exactly it is the JVM vendor dependent hence in multithreading examples we can't expect exact execution order and exact output.
- The following are various possible outputs for the above program.

p1	p2	p3	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
child thread	child thread	child thread	
child thread	child thread	child thread	------------------------
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	

Case 2: Difference between t.start() and t.run() methods.

- In the case of t.start() a new Thread will be created which is responsible for the execution of run() method.
- But in the case of t.run() no new Thread will be created and run() method will be executed just like a normal method by the main Thread.
- In the above program if we are replacing t.start() with t.run() the following is the output.

```
Output:
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
```

Entire output produced by only main Thread.

Case 3: importance of Thread class start() method.

For every Thread the required mandatory activities like registering the Thread with Thread Scheduler will takes care by Thread class start() method and programmer is responsible just to define the job of the Thread inside run() method.
That is start() method acts as best assistant to the programmer.

```
Example:
start()
{
       1. Register Thread with Thread Scheduler
       2. All other mandatory low level activities.
       3. Invoke or calling run() method.
}
```

We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java. Due to this start() is considered as **heart of multithreading**.

Case 4: If we are not overriding run() method:

If we are not overriding run() method then Thread class run() method will be executed which has empty implementation and hence we won't get any output.

```
Example:
class MyThread extends Thread
{}
class ThreadDemo
{
       public static void main(String[] args)
       {
              MyThread t=new MyThread();
              t.start();
       }
}
```

It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

Case 5: Overloding of run() method.

We can overload run() method but Thread class start() method always invokes no argument run() method the other overload run() methods we have to call explicitly then only it will be executed just like normal method.

```
Example:
class MyThread extends Thread
{
       public void run()
       {
              System.out.println("no arg method");
       }
       public void run(int i)
       {
              System.out.println("int arg method");
       }
}
class ThreadDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
        }
}
Output:
No arg method
```

Case 6: overriding of start() method:

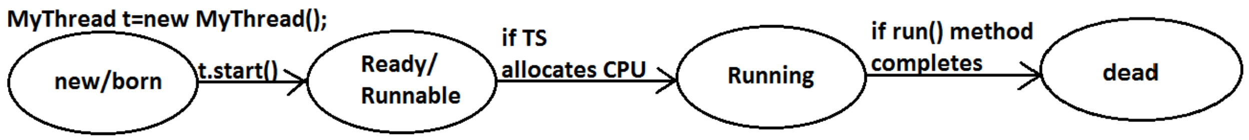
If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.

```
Example:
class MyThread extends Thread
{
        public void start()
        {
                System.out.println("start method");
        }
        public void run()
        {
                System.out.println("run method");
        }
}
class ThreadDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                System.out.println("main method");
        }
}
Output:
start method
main method
```

Entire output produced by only main Thread.
Note : It is never recommended to override start() method.

Case 7: life cycle of the Thread:

Diagram:

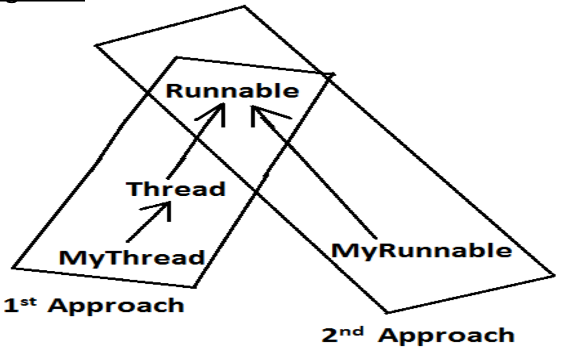


- Once we created a Thread object then the Thread is said to be in new state or born state.
- Once we call start() method then the Thread will be entered into Ready or Runnable state.
- If Thread Scheduler allocates CPU then the Thread will be entered into running state.
- Once run() method completes then the Thread will entered into dead state.

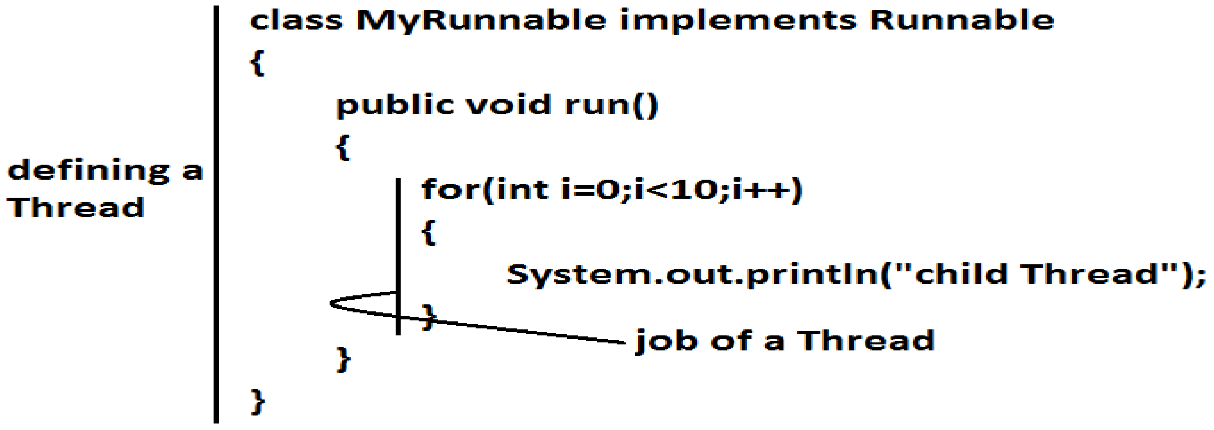
Defining a Thread by implementing Runnable interface:

We can define a Thread even by implementing Runnable interface also.
Runnable interface present in java.lang.pkg and contains only one method run().

Diagram:



Example:



```
class MyRunnable implements Runnable
{
	public void run()
	{
		for(int i=0;i<10;i++)
		{
			System.out.println("child Thread");
		}
	}
}
```

defining a Thread
job of a Thread

```
class ThreadDemo
{
	public static void main(String[] args)
	{
		MyRunnable r=new MyRunnable();
		Thread t=new Thread(r);//here r is a Target Runnable
		t.start();

		for(int i=0;i<10;i++)
		{
			System.out.println("main thread");
		}
	}
}
```

```
Output:
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
```

We can't expect exact output but there are several possible outputs.

Case study:

```
MyRunnable r=new MyRunnable();
Thread t1=new Thread();
Thread t2=new Thread(r);
```

Case 1: t1.start():

A new Thread will be created which is responsible for the execution of Thread class run()method.

```
Output:
main thread
main thread
main thread
main thread
main thread
```

Case 2: t1.run():

No new Thread will be created but Thread class run() method will be executed just like a normal method call.

```
Output:
main thread
main thread
main thread
main thread
main thread
```

Case 3: t2.start():

New Thread will be created which is responsible for the execution of MyRunnable run() method.

```
Output:
main thread
main thread
main thread
main thread
main thread
child Thread
child Thread
child Thread
child Thread
child Thread
```

Case 4: t2.run():

No new Thread will be created and MyRunnable run() method will be executed just like a normal method call.

```
Output:
child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread
```

Case 5: r.start():

We will get compile time error saying start()method is not available in MyRunnable class.

```
Output:
Compile time error
E:\SCJP>javac ThreadDemo.java
ThreadDemo.java:18: cannot find symbol
Symbol: method start()
Location: class MyRunnable
```

Case 6: r.run():

No new Thread will be created and MyRunnable class run() method will be executed just like a normal method call.

```
Output:
child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread
```

In which of the above cases a new Thread will be created which is responsible for the execution of MyRunnable run() method ?
t2.start();

In which of the above cases a new Thread will be created ?
t1.start();
t2.start();

In which of the above cases MyRunnable class run() will be executed ?
t2.start();
t2.run();
r.run();

<u>Best approach to define a Thread:</u>

- Among the 2 ways of defining a Thread, implements Runnable approach is always recommended.
- In the 1st approach our class should always extends Thread class there is no chance of extending any other class hence we are missing the benefits of inheritance.
- But in the 2nd approach while implementing Runnable interface we can extend some other class also. Hence implements Runnable mechanism is recommended to define a Thread.

Thread class constructors:

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r,String name);
5. Thread t=new Thread(ThreadGroup g,String name);

6. Thread t=new Thread(ThreadGroup g,Runnable r);
7. Thread t=new Thread(ThreadGroup g,Runnable r,String name);
8. Thread t=new Thread(ThreadGroup g,Runnable r,String name,long stackSize);

Thread Priorities

- Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.
- The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least priority and 10 is highest priority.
- Thread class defines the following constants to represent some standard priorities.
    1. Thread. MIN_PRIORITY----------1
    2. Thread. MAX_PRIORITY----------10
    3. Thread. NORM_PRIORITY--------5
- There are no constants like Thread.LOW_PRIORITY, Thread.HIGH_PRIORITY
- Thread scheduler uses these priorities while allocating CPU.
- The Thread which is having highest priority will get chance for 1st execution.
- If 2 Threads having the same priority then we can't expect exact execution order it depends on Thread scheduler whose behavior is vendor dependent.
- We can get and set the priority of a Thread by using the following methods.
    1. public final int getPriority()
    2. public final void setPriority(int newPriority);//the allowed values are 1 to 10
- The allowed values are 1 to 10 otherwise we will get runtime exception saying "IllegalArgumentException".

Default priority:

The default priority only for the main Thread is 5. But for all the remaining Threads the default priority will be inheriting from parent to child. That is whatever the priority parent has by default the same priority will be for the child also.

Example 1:

```
class MyThread extends Thread
{}
class ThreadPriorityDemo
{
        public static void main(String[] args)
        {
                System.out.println(Thread.currentThread().getPriority());//5
                Thread.currentThread().setPriority(9);
                MyThread t=new MyThread();
                System.out.println(t.getPriority());//9
        }
}
```

Example 2:

```
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<10;i++)
                {
                        System.out.println("child thread");
                }
        }
}
class ThreadPriorityDemo
```

```
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                //t.setPriority(10);        //----> 1
                t.start();
                for(int i=0;i<10;i++)
                {
                        System.out.println("main thread");
                }
        }
}
```

- If we are commenting line 1 then both main and child Threads will have the same priority and hence we can't expect exact execution order.
- If we are not commenting line 1 then child Thread has the priority 10 and main Thread has the priority 5 hence child Thread will get chance for execution and after completing child Thread main Thread will get the chance in this the output is:

Output:

```
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
```

Some operating systems(like windowsXP) may not provide proper support for Thread priorities. We have to install separate bats provided by vendor to provide support for priorities.

The Methods to Prevent a Thread from Execution:

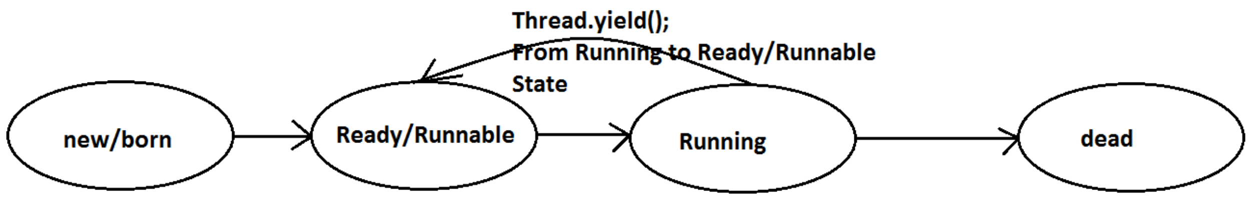
We can prevent(stop) a Thread execution by using the following methods.

1. yield();
2. join();
3. sleep();

yield():

1. yield() method causes "to pause current executing Thread for giving the chance of remaining waiting Threads of same priority".
2. If all waiting Threads have the low priority or if there is no waiting Threads then the same Thread will be continued its execution.
3. If several waiting Threads with same priority available then we can't expect exact which Thread will get chance for execution.
4. The Thread which is yielded when it get chance once again for execution is depends on mercy of the Thread scheduler.
5. public static native void yield();

Diagram:



```
Example:
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<5;i++)
		{
			Thread.yield();
			System.out.println("child thread");
		}
	}
}
class ThreadYieldDemo
{
	public static void main(String[] args)
	{
		MyThread t=new MyThread();
```

```
                t.start();
                for(int i=0;i<5;i++)
                {
                        System.out.println("main thread");
                }
        }
}
Output:
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
```

In the above program child Thread always calling yield() method and hence main Thread will get the chance more number of times for execution.
Hence the chance of completing the main Thread first is high.

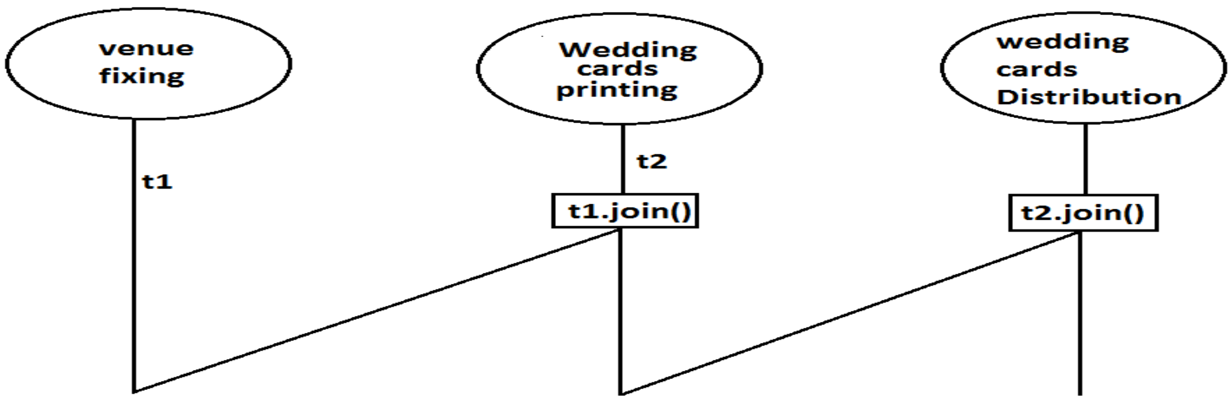
Note : Some operating systems may not provide proper support for yield() method.

Join( ):

If a Thread wants to wait until completing some other Thread then we should go for join() method.

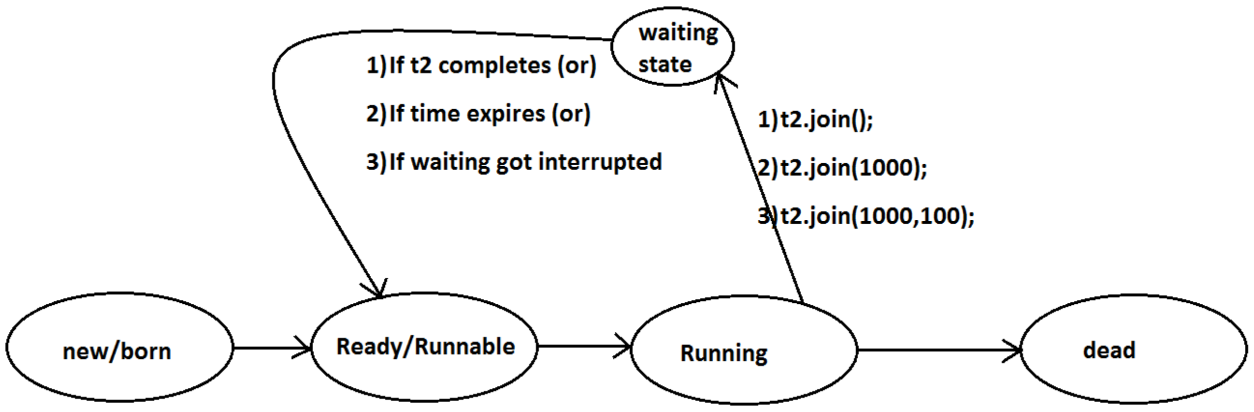
Example: If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.

Diagram:



1. public final void join()throws InterruptedException
2. public final void join(long ms) throws InterruptedException
3. public final void join(long ms,int ns) throws InterruptedException

Diagram:



Every join() method throws InterruptedException, which is checked exception hence compulsory we should handle either by **try catch** or by **throws** keyword.
Otherwise we will get compiletime error.

```
Example:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("Sita Thread");
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch (InterruptedException e){}
                }
        }
}
class ThreadJoinDemo
{
        public static void main(String[] args)throws
InterruptedException
        {
                MyThread t=new MyThread();
                t.start();
                //t.join();   //--->1
                for(int i=0;i<5;i++)
                {
                        System.out.println("Rama Thread");
                }
        }
}
```

- If we are commenting line 1 then both Threads will be executed simultaneously and we can't expect exact execution order.
- If we are not commenting line 1 then main Thread will wait until completing child Thread in this the output is sita Thread 5 times followed by Rama Thread 5 times.

Waiting of child Thread untill completing main Thread :

```
Example:
class MyThread extends Thread
{
 static Thread mt;
        public void run()
        {
                try
                {
                        mt.join();
                }
                catch (InterruptedException e){}


                for(int i=0;i<5;i++)
                {
                        System.out.println("Child Thread");
                }
        }
}
class ThreadJoinDemo
{
        public static void main(String[] args)throws
InterruptedException
        {
                MyThread mt=Thread.currentThread();
                MyThread t=new MyThread();
                t.start();

                for(int i=0;i<5;i++)
                {
                        Thread.sleep(2000);
                        System.out.println("Main Thread");
                }
        }
}

Output :
```

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
```

Note :
If main thread calls join() on child thread object and child thread called join() on main thread object then both threads will wait for each other forever and the program will be hanged(like deadlock if a Thread class join() method on the same thread itself then the program will be hanged).

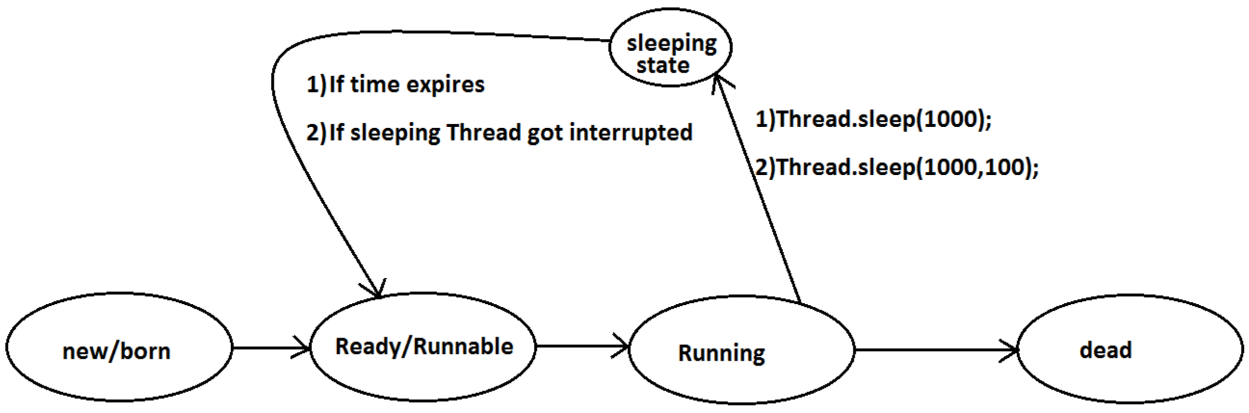
```
Example :

class ThreadDemo
{
       public static void main() throws InterruptedException
       {
              Thread.currentThread().join();
              ---------------   --------
               main               main
       }

}
```

Sleep() method:

If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.

1. **public static native void sleep(long ms) throws InterruptedException**
2. **public static void sleep(long ms,int ns)throws InterruptedException**

Diagram:



```
Example:
class ThreadJoinDemo
{
        public static void main(String[] args)throws
InterruptedException
        {
                System.out.println("M");
                Thread.sleep(3000);
                System.out.println("E");
                Thread.sleep(3000);
                System.out.println("G");
                Thread.sleep(3000);
                System.out.println("A");
        }
}
Output:
M
E
G
A
```

Interrupting a Thread:

How a Thread can interrupt another thread ?

If a Thread can interrupt a sleeping or waiting Thread by using interrupt()(break off) method of Thread class.

public void interrupt();

```
Example:
class MyThread extends Thread
{
        public void run()
        {
                try
                {
                        for(int i=0;i<5;i++)
                        {
                                System.out.println("i am lazy Thread :"+i);
                                Thread.sleep(2000);
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("i got interrupted");
                }
        }
}
class ThreadInterruptDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                //t.interrupt();     //--->1
                System.out.println("end of main thread");
        }
}
```

- If we are commenting line 1 then main Thread won't interrupt child Thread and hence child Thread will be continued until its completion.
- If we are not commenting line 1 then main Thread interrupts child Thread and hence child Thread won't continued until its completion in this case the output is:

```
End of main thread
I am lazy Thread: 0
I got interrupted
```

Note:

- Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.
- If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target Thread entered into sleeping or waiting state it will effect immediately.
- In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.

```
Example:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("iam lazy thread");
                }
                System.out.println("I'm entered into sleeping stage");
              try
              {
                     Thread.sleep(3000);
              }
              catch (InterruptedException e)
              {
                     System.out.println("i got interrupted");
              }
       }
}
class ThreadInterruptDemo1
{
       public static void main(String[] args)
       {
              MyThread t=new MyThread();
              t.start();
              t.interrupt();
              System.out.println("end of main thread");
       }
}
```

- In the above program interrupt() method call invoked by main Thread will wait until child Thread entered into sleeping state.
- Once child Thread entered into sleeping state then it will be interrupted immediately.

Compression of yield, join and sleep() method?

property	Yield( )	Join( )	Sleep( )
1) Purpose?	To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.	If a Thread wants to wait until completing some other Thread then we should go for join.	If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep( ) method.
2) Is it static?	yes	no	yes
3) Is it final?	no	yes	no
4) Is it overloaded?	No	yes	yes
5) Is it throws InterruptedException?	no	yes	yes
6) Is it native method?	yes	no	sleep(long ms) -->native sleep(long ms,int ns) -->non-native

LIFECYCLE OF THREAD:

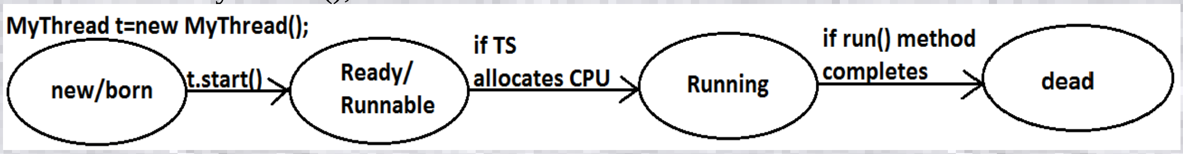
Life cycle stages are:-

1) New
2) Ready
3) Running state
4) Blocked / waiting / non-running mode
5) Dead state

New :-

Once we created a Thread object then the Thread is said to be in new state or born state.

MyThread t=new MyThread();



Ready :-

Once we call start() method then the Thread will be entered into Ready or Runnable state.

t.start()

Running state:-

If thread scheduler allocates CPU for particular thread. Thread goes to running state. The Thread is running state means the run() is executed.

Blocked State:-

If the running thread got interrupted of goes to sleeping state at that moment it goes to the blocked state.

Dead State:-

Once run() method completes then the Thread will entered into dead state. If the business logic of the project is completed means run() over thread goes dead state.

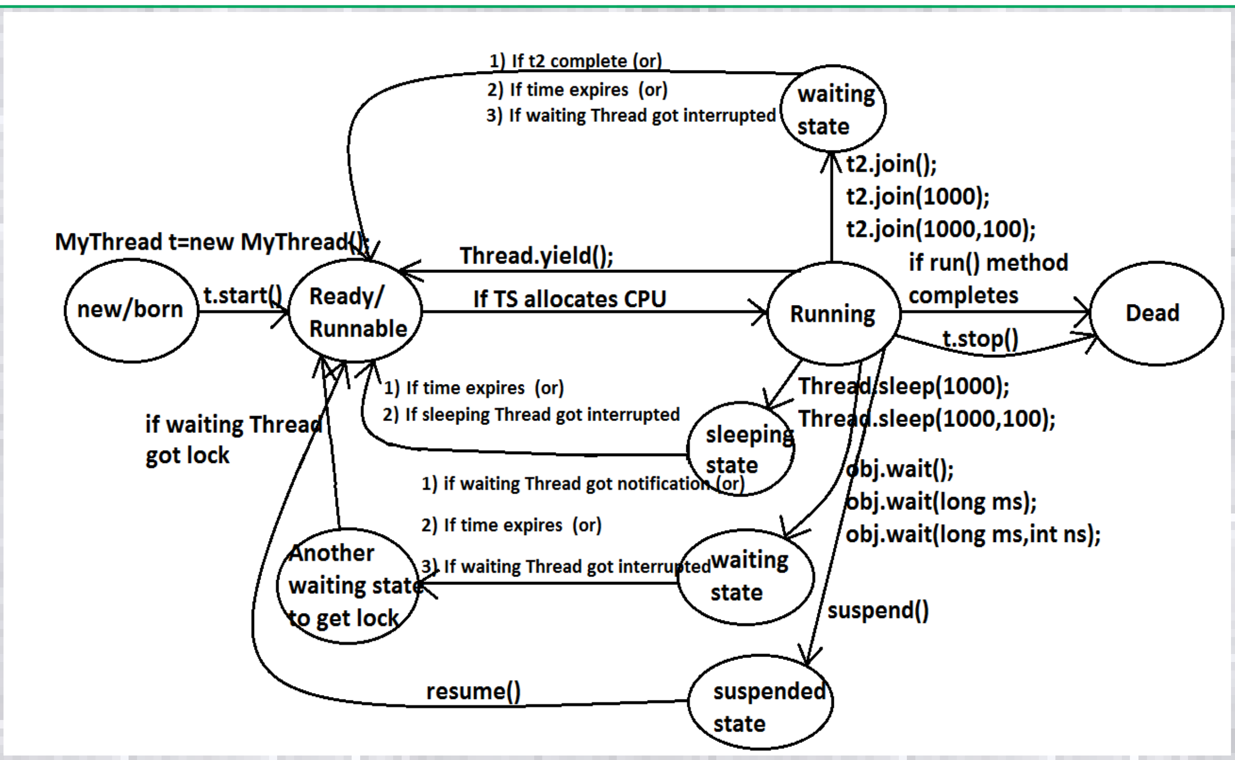


Figure: LIFE CYCLE OF A THREAD

Synchronization

1. **Synchronized** is the keyword applicable for **methods** and **blocks** but not for classes and variables.
2. If a method or block declared as the synchronized then at a time **only one Thread is allowed** to execute that method or block on the given object.
3. The main advantage of synchronized keyword is we can resolve **data inconsistency** problems.
4. But the main disadvantage of synchronized keyword is it **increases waiting time** of the Thread and effects performance of the system.
5. Hence if there is no specific requirement then never recommended to use synchronized keyword.
6. Internally synchronization concept is implemented by using **lock concept**.
7. Every java object has a lock. A lock has only one key. Most of the time lock is unlocked and nobody cares.
8. If a Thread wants to execute any synchronized method on the given object, First it has to get the lock of that object. Once a Thread got the lock of that object then it is allowed to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
9. While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented **based on object** but not based on method].

```
Example:
class Display
{
	public synchronized void wish(String name)
	{
		for(int i=0;i<5;i++)
		{
			System.out.print("good morning:");
			try
			{
				Thread.sleep(1000);
			}
			catch (InterruptedException e)
			{}
			System.out.println(name);
		}
	}
}
class MyThread extends Thread
{
	Display d;
	String name;
	MyThread(Display d,String name)
	{
```

```
                this.d=d;
                this.name=name;
        }
        public void run()
        {
                d.wish(name);
        }
}
class SynchronizedDemo
{
        public static void main(String[] args)
        {
                Display d1=new Display();
                MyThread t1=new MyThread(d1,"dhoni");
                MyThread t2=new MyThread(d1,"yuvaraj");
                t1.start();
                t2.start();
        }
}
```

If **we are not declaring wish( ) method as synchronized** then both Threads will be executed simultaneously and we will get irregular output.

```
Output:
good morning:good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
dhoni
```

If **we declare wish( ) method as synchronized** then the Threads will be executed one by one that is until completing the 1st Thread the 2nd Thread will wait in this case we will get regular output which is nothing but

```
Output:
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
```

Example :-

```
class Test
{
	public static synchronized void x(String msg)//only one thread is able to access
	{
		try
		{
			System.out.println(msg);
			Thread.sleep(4000);
			System.out.println(msg);
			Thread.sleep(4000);
		}
		catch(Exception e)
		{	e.printStackTrace();
		}
	}
}
class MyThread1 extends Thread
{
	public void run(){Test.x("ratan");}
}
class MyThread2 extends Thread
{
	public void run(){Test.x("anu");}
}
class MyThread3 extends Thread
{
	public void run(){Test.x("banu");}
}
class TestDemo
{
	public static void main(String[] args)//main thread -1
	{
		MyThread1 t1 = new MyThread1();
		MyThread2 t2 = new MyThread2();
		MyThread3 t3 = new MyThread3();
		t1.start();//2-Threads
		t2.start();//3-Threads
		t3.start();//4-Threads
	}
}
```

If method is synchronized:

```
D:\DP>java ThreadDemo
anu
anu
banu
banu
ratan
ratan
```

if method is non-synchronized:-

```
D:\DP>java ThreadDemo
banu
```

```
ratan
anu
banu
anu
ratan
```

synchronized blocks:-

If the application method contains 100 lines but if we want to synchronized only 10 lines of code use synchronized blocks.
The synchronized block contains less scope compare to method.
If we are writing all the method code inside the synchronized blocks it will work same as the synchronized method.

Syntax:-

```
synchronized(object)
{
	//code
}
class Heroin
{
	public void message(String msg)
	{
		synchronized(this){
			System.out.println("hi "+msg+" "+Thread.currentThread().getName());
			try{Thread.sleep(5000);}
			catch(InterruptedException e){e.printStackTrace();}
		}
		System.out.println("hi krishnasoft");
	}
}
class MyThread1 extends Thread
{
	Heroin h;
	MyThread1(Heroin h)
	{this.h=h;
	}
	public void run()
	{
		h.message("Samantha");
	}
}
class MyThread2 extends Thread
{
	Heroin h;
	MyThread2(Heroin h)
	{this.h=h;
	}
	public void run()
	{
		h.message("Kavi");
	}
}
class ThreadDemo
{
	public static void main(String[] args)
	{
		Heroin h = new Heroin();
		MyThread1 t1 = new MyThread1(h);
		MyThread2 t2 = new MyThread2(h);
		t1.start();
		t2.start();
	}
}
```

Deadlock:

- If two Threads are waiting for each other forever (without end) such type of situation **(infinite waiting)** is called deadlock.
- There are no resolution techniques for dead lock but several prevention (avoidance) techniques are possible.
- Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword we have to take special care.

```
Example:
class A
{
        public synchronized void tata(B b)
        {
                System.out.println("Thread1 starts execution of tata() method");
                try
                {
                        Thread.sleep(2000);
                }
                catch (InterruptedException e)
                {}
                System.out.println("Thread1 trying to call b.last()");
                b.last();
        }
        public synchronized void last()
        {
                System.out.println("inside A, this is last()method");
        }
}
class B
{
        public synchronized void toto(A a)
        {
        System.out.println("Thread2 starts execution of toto() method");
        try
        {
                Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {}
        System.out.println("Thread2 trying to call a.last()");
        a.last();
        }
        public synchronized void last()
        {
        System.out.println("inside B, this is last() method");
        }
}
class DeadLock implements Runnable
{
        A a=new A();
        B b=new B();
        DeadLock()
        {
                Thread t=new Thread(this);
                t.start();
                a.tata(b);//main thread
        }
        public void run()
        {
                b.toto(a);//child thread
        }
        public static void main(String[] args)
        {
                new DeadLock();//main thread
        }
}
Output:
Thread1 starts execution of tata() method
Thread2 starts execution of toto() method
Thread2 trying to call a.last()
Thread1 trying to call b.last()
//here cursor always waiting.
```

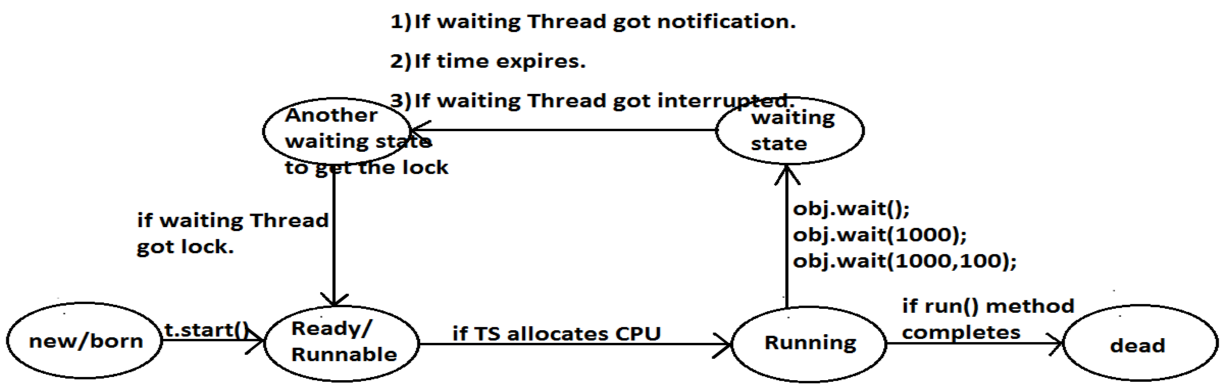
Note : If we remove atleast one synchronized keyword then we won't get DeadLock. Hence synchronized keyword in the only reason for DeadLock due to this while using synchronized keyword we have to handle carefully.

Inter Thread communication (wait( ),notify( ), notifyAll( )):

- Two Threads can communicate with each other by using **wait( ), notify( ) and notifyAll( )** methods.
- wait(), notify() and notifyAll() methods are available in **Object class** but not in Thread class because Thread can call these methods on any common object.
- To call wait(), notify() and notifyAll() methods compulsory the current Thread should be owner of that object
  i.e., current Thread should has lock of that object
  i.e., current Thread should be in synchronized area. Hence we can call wait(), notify() and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying**IllegalMonitorStateException**.
- Once a Thread calls wait() method on the given object, First it releases the lock of that object **immediately** and entered into waiting state.
- Once a Thread calls notify() (or) notifyAll() methods it releases the lock of that object but **may not immediately.**
- Except these (wait( ),notify( ),notifyAll( )) methods there is no other place(method) where the lock release will be happen.

Method	Is Thread Releases Lock?
yield( )	No
join( )	No
sleep()	No
wait( )	Yes
notify( )	Yes
notifyAll( )	Yes

- Once a Thread calls wait(), notify(), notifyAll() methods on any object then it releases the lock of that particular object **but not all locks it has.**
  1. public final void wait( )throws InterruptedException
  2. public final native void wait(long ms)throws InterruptedException
  3. public final void wait(long ms,int ns)throws InterruptedException
  4. public final native void notify()
  5. public final void notifyAll()


1)If waiting Thread got notification.
2)If time expires.
3)If waiting Thread got interrupted.
Another waiting state to get the lock
waiting state
if waiting Thread got lock.
obj.wait();
obj.wait(1000);
obj.wait(1000,100);
new/born
t.start()
Ready/ Runnable
if TS allocates CPU
Running
if run() method completes
dead

```
Example 1:
class ThreadA
{
        public static void main(String[] args)throws InterruptedException
        {
        ThreadB b=new ThreadB();
        b.start();
        synchronized(b)
        {
        System.out.println("main Thread calling wait() method");//step-1
        b.wait();
        System.out.println("main Thread got notification call");//step-4
        System.out.println(b.total);
        }
        }
}
class ThreadB extends Thread
{
        int total=0;
        public void run()
        {
        synchronized(this)
        {
        System.out.println("child thread starts calcuation");//step-2
                for(int i=0;i<=100;i++)
                {
                        total=total+i;
                }
        System.out.println("child thread giving notification call");//step-3
        this.notify();
        }
        }
}
Output:
main Thread calling wait() method
child thread starts calculation
child thread giving notification call
main Thread got notification call
5050
```

Notify vs notifyAll():

- We can use notify( ) method to give notification **for only one Thread**. If multiple Threads are waiting then only one Thread will get the chance and remaining Threads has to wait for further notification. But which Thread will be notify(inform) we can't expect exactly it depends on JVM.
- We can use notifyAll( ) method to give the notification **for all waiting Threads**. All waiting Threads will be notified and will be executed one by one, because they are required lock

Note: On which object we are calling wait( ), notify( ) and notifyAll( ) methods that corresponding object lock we have to get but not other object locks.

Which of the folowing statements are True ?

1. Once a Thread calls wait() on any Object immediately it will entered into waiting state without releasing the lock ?
   NO
2. Once a Thread calls wait() on any Object it reduces the lock of that Object but may not immediately ?
   NO
3. Once a Thread calls wait() on any Object it immediately releases all locks whatever it has and entered into waiting state ?
   NO
4. Once a Thread calls wait() on any Object it immediately releases the lock of that perticular Object and entered into waiting state ?
   YES
5. Once a Thread calls notify() on any Object it immediately releases the lock of that Object ?
   NO
6. Once a Thread calls notify() on any Object it releases the lock of that Object but may not immediately ?
   YES

Daemon Threads:

The Threads which are **executing in the background** are called daemon Threads. The main objective of daemon Threads is to provide support for non-daemon Threads like **main Thread.**

Example:
Garbage collector

When ever the program runs with low memory, the JVM will execute Garbage Collector to provide **free memory**. So that the main Thread can continue it's execution.

- We can check whether the Thread is daemon or not by using **isDaemon() method** of Thread class.
  public final boolean isDaemon();
- We can change daemon nature of a Thread by using **setDaemon()** method.
  public final void setDaemon(boolean b);
- But we can change daemon nature **before starting Thread only**. That is after starting the Thread if we are trying to change the daemon nature we will get R.E saying ***IllegalThreadStateException***.
- **Default Nature :** Main Thread is always non daemon and we can't change its daemon nature because **it's already started at the beginning only.**
- Main Thread is always non daemon and for the remaining Threads **daemon nature will be inheriting from parent to child** that is if the parent is daemon, child is also daemon and if the parent is non daemon then child is also non daemon.
- Whenever the last non daemon Thread terminates automatically all daemon Threads will be terminated.

```
Example:
class MyThread extends Thread
{ }
class DaemonThreadDemo
{
        public static void main(String[] args)
        {
        System.out.println(Thread.currentThread().isDaemon());
        MyThread t=new MyThread();
        System.out.println(t.isDaemon());                1
        t.start();
        t.setDaemon(true);
        System.out.println(t.isDaemon());
        }
}
Output:
false
false
RE:IllegalThreadStateException
```

```
Example:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<10;i++)
                {
                        System.out.println("lazy thread");
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch (InterruptedException e)
                        {}
                }
        }
}
class DaemonThreadDemo
{
        public static void main(String[] args)
        {
        MyThread t=new MyThread();
        t.setDaemon(true);     //-->1
        t.start();
        System.out.println("end of main Thread");
        }
}
Output:
End of main Thread
```

Deadlock vs Starvation:

- A long waiting of a Thread which never ends is called **deadlock**.
- A long waiting of a Thread which ends at certain point is called **starvation**.
- A low priority Thread has to wait until completing all high priority Threads.
- This long waiting of Thread which ends at certain point is called **starvation.**

How to kill a Thread in the middle of the line?

- We can call **stop()** method to stop a Thread in the middle then it will be entered into dead state immediately.
  public final void stop();
- stop() method has been deprecated and hence not recommended to use.

suspend and resume methods:

- A Thread can suspend another Thread by using suspend() method then that Thread will be paused temporarily.
- A Thread can resume a suspended Thread by using resume() method then suspended Thread will continue its execution.
  1. **public final void suspend();**

2. **public final void resume();**

- Both methods are deprecated and not recommended to use.

RACE condition:

Executing multiple Threads simultaneously and causing data inconsistency problems is nothing but **Race condition**
we can resolve race condition by using synchronized keyword.

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods are given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	int	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	int	getMaxPriority()	This method returns the maximum priority of

			the thread group.
7)	String	getName()	This method returns the name of the thread group.
8)	ThreadGroup	getParent()	This method returns the parent of the thread group.
9)	void	interrupt()	This method interrupts all threads in the thread group.
10)	boolean	isDaemon()	This method tests if the thread group is a daemon thread group.
11)	void	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	boolean	isDestroyed()	This method tests if this thread group has been destroyed.
13)	void	list()	This method prints information about the thread group to the standard output.
14)	boolean	parentOf(ThreadGroup g	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	suspend()	This method is used to suspend all threads in the thread group.
16)	void	resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.
17)	void	setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void	stop()	This method is used to stop all threads in the thread group.
19)	String	toString()	This method returns a string representation of the Thread group.

Let's see a code to group multiple threads.

1. ThreadGroup tg1 = new ThreadGroup("Group A");
2. Thread t1 = new Thread(tg1,new MyRunnable(),"one");
3. Thread t2 = new Thread(tg1,new MyRunnable(),"two");
4. Thread t3 = new Thread(tg1,new MyRunnable(),"three");

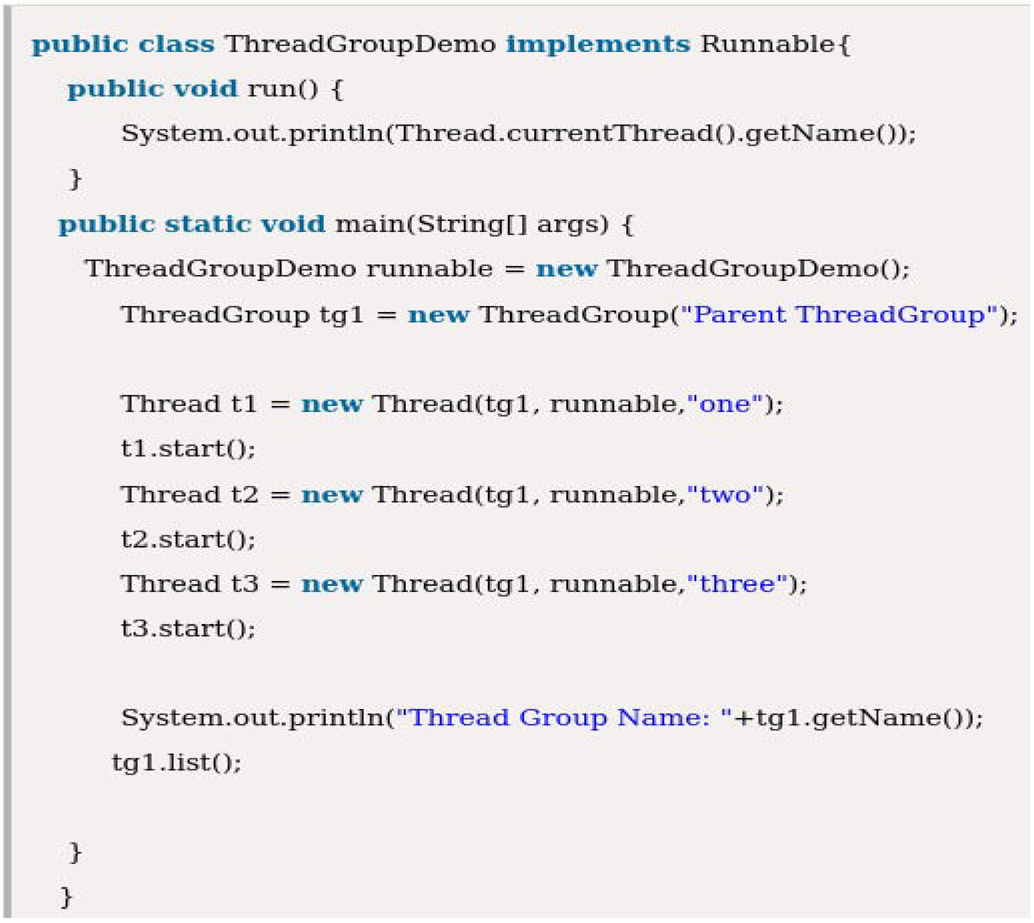
Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

1. Thread.currentThread().getThreadGroup().interrupt();

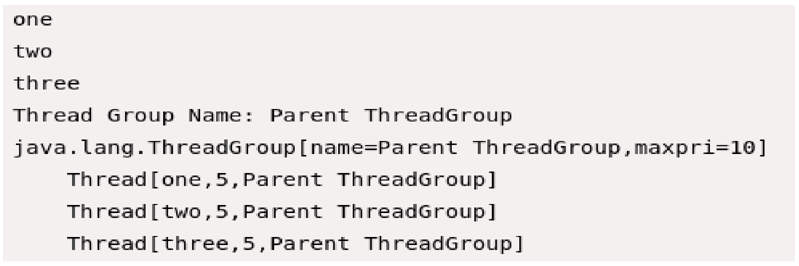
ThreadGroup Example

File: ThreadGroupDemo.java



```
public class ThreadGroupDemo implements Runnable{
   public void run() {
        System.out.println(Thread.currentThread().getName());
   }
  public static void main(String[] args) {
    ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

        Thread t1 = new Thread(tg1, runnable,"one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable,"two");
        t2.start();
        Thread t3 = new Thread(tg1, runnable,"three");
        t3.start();

        System.out.println("Thread Group Name: "+tg1.getName());
       tg1.list();

   }
  }
```

Output:



```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
    Thread[one,5,Parent ThreadGroup]
    Thread[two,5,Parent ThreadGroup]
    Thread[three,5,Parent ThreadGroup]
```