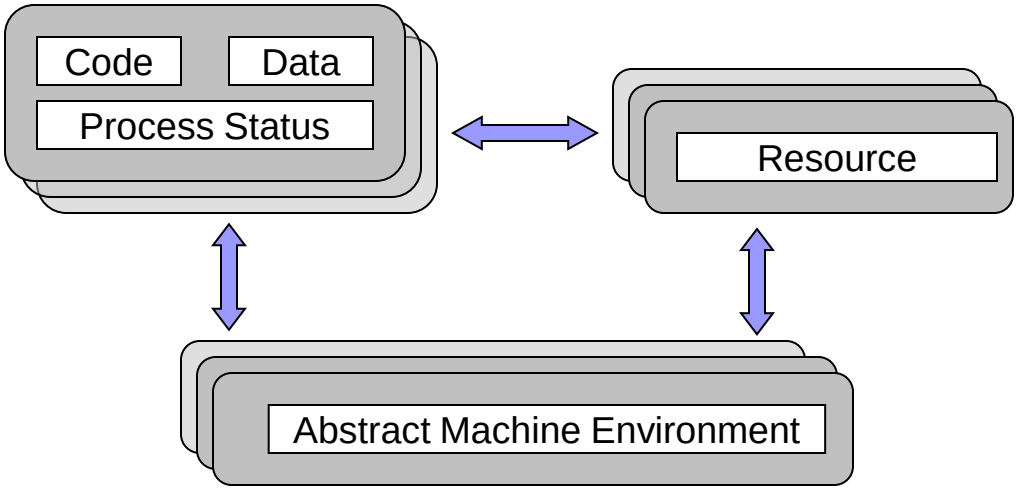
Multi-Threading in java

Outline

- Process versus Thread
- Synchronization
- Multithreading with Java

Process Model

- A **process** is a sequential program in execution.
- A **process** is a unit of computation.
- **Process components:**
  - The program (code) to be executed.
  - The data on which the program will execute.
  - Resources required by the program.
  - The status of the process execution.
- **A process runs in an abstract machine environment (could be OS) that manages the sharing and isolation of resources among the community of processes.**



Program and Process

Program and process – distinction?

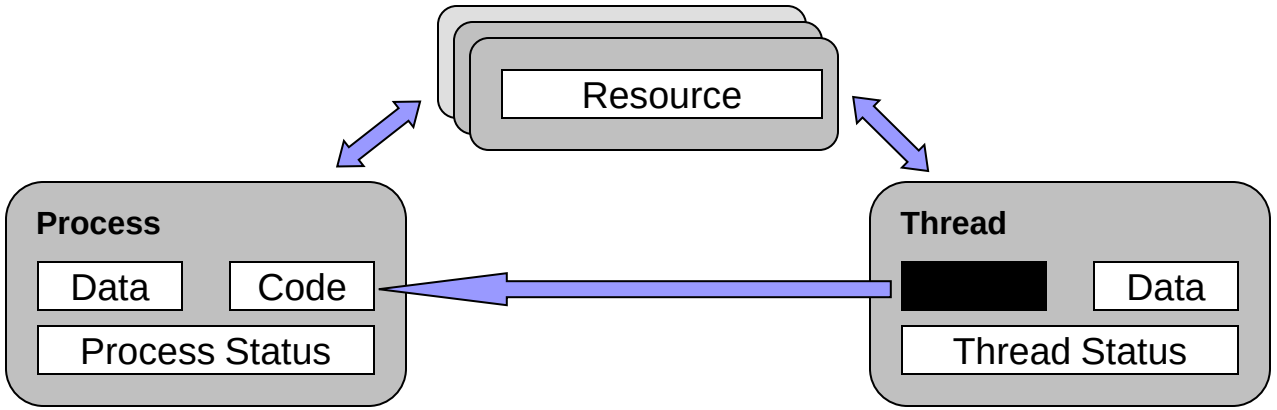
- A program is a **static entity** made up of program statements. The latter define the run-time behavior.

- A process is a **dynamic entity** that executes a program on a particular set of data.

- **Two or more processes could execute the same program**, each using their own data and resources.

Thread Model

- **A thread** is an alternative form (to the process) of schedulable unit of computation.
- In the **thread model:**
  - Each thread is associated with a process.
  - A thread is an entity that executes by **relying on the code and resources, holding by the associated process**.
  - Several threads could be associated with a single process. Those threads share the code and resources of the process.
  - A thread allocates part of the process's resources for its needs.
  - A thread has its own data and status.



Introduction: Difference between process and Thread

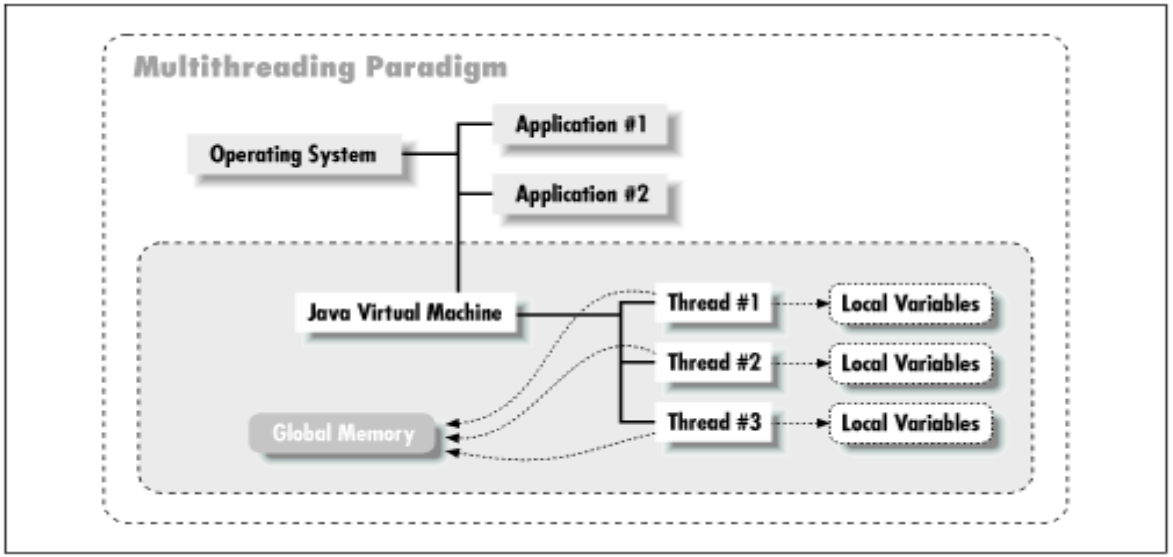
S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Introduction: Threads

Non-Daemon Threads (User Thread)	Daemon Thread
High priority threads.	Low priority threads.
Always run on foreground	Run on background
Designed to do some specific task.	Designed to support the user threads.
Created by the user.	Mostly created by the JVM (Ex: Garbage collector). User can also create Daemon Threads
JVM wait until user threads to finish their work. It never exit until all user threads finish their work.	If all user threads have finished their work JVM will force the daemon threads to terminate

Multithreading

- **A program with multiple threads** running within a single instance **could be considered** as a multitasking system within an OS.
- In a multithreading program, threads have the following properties:
  - A thread begin execution at a predefined, well-known location. For one of the threads in the program, that location is the ***main()*** method; for the rest of the threads, it is a **particular location** the programmer decides on when the code is written.
  - A thread executes code in an **ordered, predefined sequence**.
  - A thread executes its code **independently** of the other threads.
  - The threads appear to have a **certain degree of simultaneous execution**.

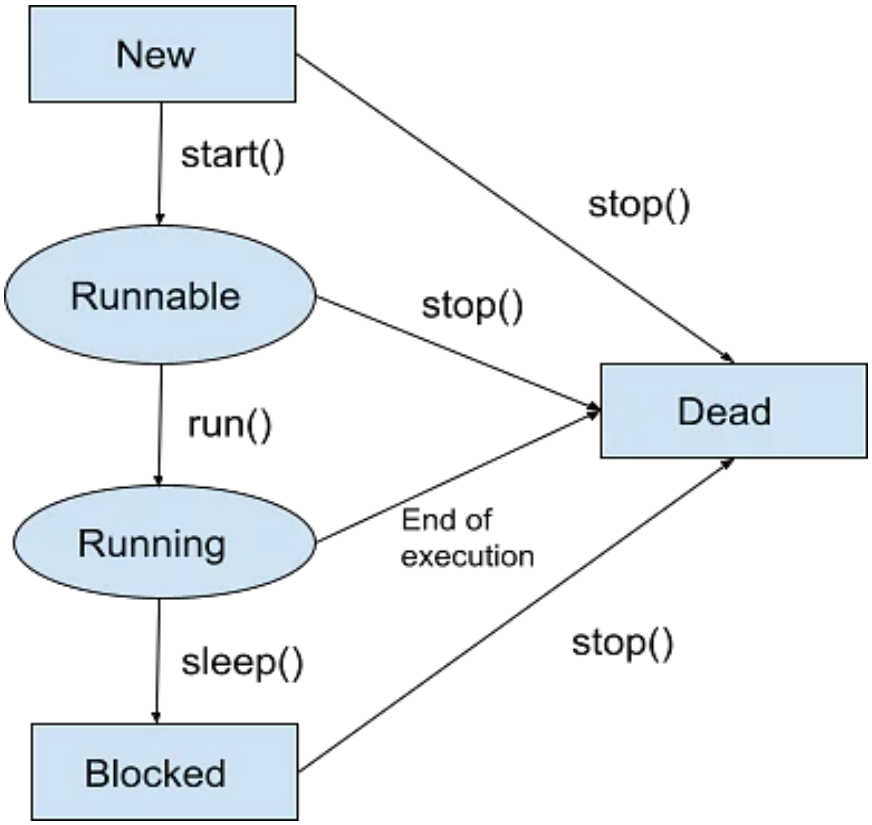


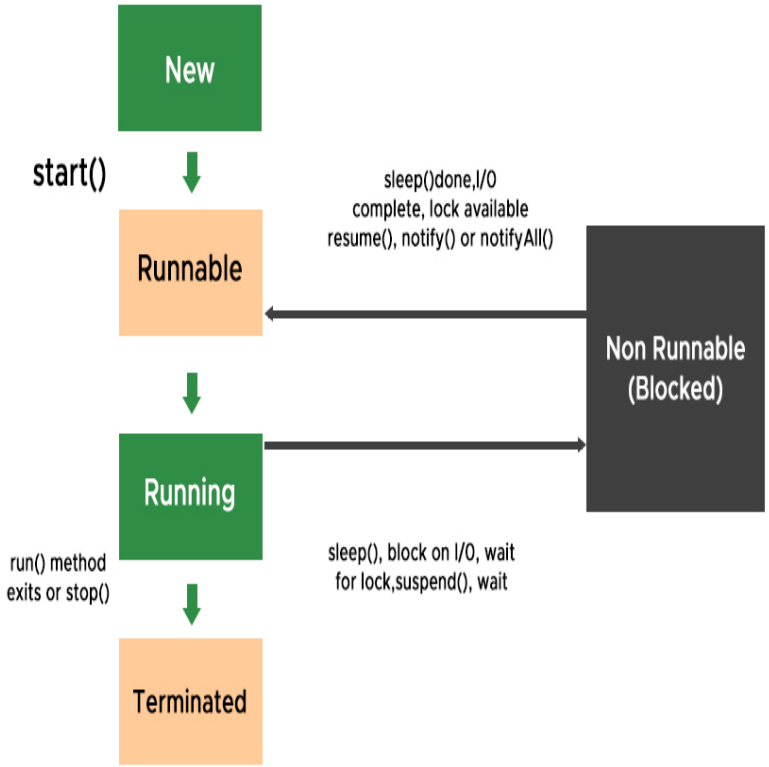
Lifecycle of a Thread in Java

The Life Cycle of a Thread in Java refers to the state transformations of a thread that begins with its birth and ends with its death. When a thread instance is generated and executed by calling the start() method of the Thread class, the thread enters the runnable state. When the sleep() or wait() methods of the Thread class are called, the thread enters a non-runnable mode. Thread returns from non-runnable state to runnable state and starts statement execution. The thread dies when it exits the run() process, as given below:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead

Lifecycle of a Thread in Java





Lifecycle of a Thread in Java

New State: As we use the Thread class to construct a thread entity, the thread is born and is defined as being in the New state. That is, when a thread is created, it enters a new state, but the start() method on the instance has not yet been invoked.

Runnable State: When a new thread's start() function is called, it enters a runnable state. The thread has entered the queue (line) of threads waiting for execution.

Running State: When a thread from the runnable state is chosen for execution by the thread scheduler, it joins the running state. This is the state in which the thread directly executes its operations. Only from the runnable state will a thread enter the running state.

Blocked State: The thread class object persists, but it cannot be selected for execution by the scheduler. It is now inactive.

Dead State: when a thread exits the run() process, it is terminated. When the stop() function is invoked, a thread will also go dead.

Java Thread Priorities

The number of services assigned to a given thread is referred to as its priority. Any thread generated in the JVM is given a priority. The priority scale runs from 1 to 10.

1 is known as the lowest priority.

5 is known as standard priority.

10 represents the highest level of priority.

The main thread's priority is set to 5 by default, and each child thread will have the same priority as its parent thread. We have the ability to adjust the priority of any thread, whether it is the main thread or a user-defined thread. It is advised to adjust the priority using the Thread class's constants, which are as follows:

1. Thread.MIN_PRIORITY;
2. Thread.NORM_PRIORITY;
3. Thread.MAX_PRIORITY;

Java Thread Priorities

```
package test;
public class ThreadPriority extends Thread
{
    public void run ()
    {
        System.out.println ("running thread priority is:" +
   Thread.currentThread ().getPriority ());

    }
    public static void main (String args[])
    {
        ThreadPriority m1 = new ThreadPriority ();
        ThreadPriority m2 = new ThreadPriority ();
        m1.setPriority (Thread.MIN_PRIORITY);
        m2.setPriority (Thread.MAX_PRIORITY);
        m1.start ();
        m2.start ();
    }
}
```

Output:
running thread priority is: 1
running thread priority is: 10

Thread class

Methods of Thread class:

start() – Starts the thread.

getState() – It returns the state of the thread.

currentThread(): Returns a reference to the currently executing thread object.

activeCount(): Returns an estimate of the number of active threads in the current thread's thread group and its subgroups.

getName() – It returns the name of the thread.

getPriority() – It returns the priority of the thread.

sleep() – Stop the thread for the specified time.

Join() – Stop the current thread until the called thread gets terminated.

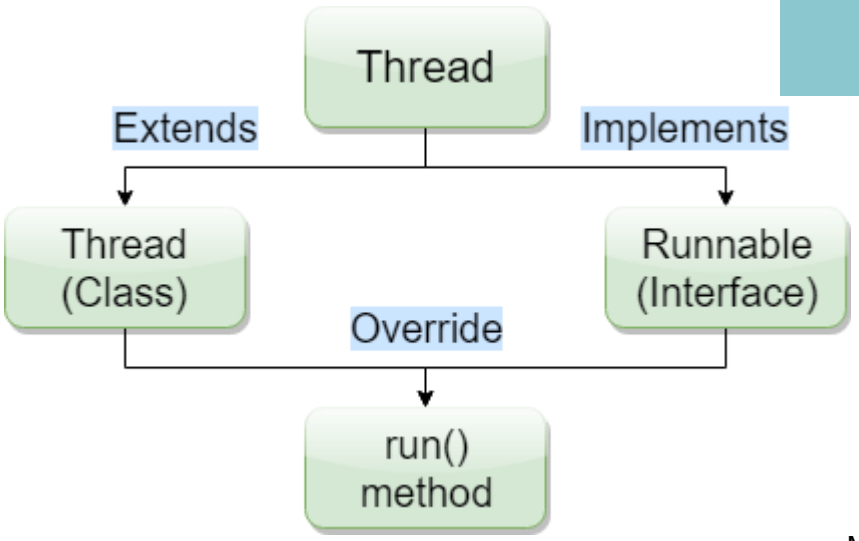
Thread class

Sr.No.	Constructor & Description
1	**Thread():This allocates a new Thread object.**
2	**Thread(Runnable target):This allocates a new Thread object.**
3	**Thread(Runnable target, String name):This allocates a new Thread object.**
4	**Thread(String name):This constructs allocates a new Thread object.**
5	**Thread(ThreadGroup group, Runnable target):This allocates a new Thread object.**
6	**Thread(ThreadGroup group, Runnable target, String name):This allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.**
7	**Thread(ThreadGroup group, Runnable target, String name, long stackSize):This allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified stack size.**
8	**Thread(ThreadGroup group, String name):This allocates a new Thread object.**

Thread class and runnable interface

Implementing Runnable Interface

- The Runnable interface has only one method
  - run() method
- The Runnable interface should be implemented by
  - Any class whose instances are intended to be executed as a thread
- The class must define run() method of no arguments
  - The run() method is like main() for the new thread
- A class that implements Runnable can run by
  - Instantiating a Thread instance and passing itself in as the target



Creation of thread

There are two ways to create a thread:

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

By implementing Runnable interface:

Runnable interface:The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.Inside the run we will define the code that constitute the thread.It can call other methods, use other classes and declare variables , jst like the main() thread. The only difference is that run() establish the entry point for another concurrent thread execution within your program. The thread will end when run() return.

Creation of thread: By implementing Runnable interface

```
class Newthread implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Newthread m1=new Newthread();
Thread t1 =new Thread(m1);   // Using the constructor
Thread(Runnable r)
t1.start();
 }
}
```

Output:

thread is running…

Creation of thread: By implementing Runnable interface

```
import java.util.Scanner;
//Creating thread implementing Runnable interface
class NewThread implements Runnable{
    Thread t;    String Name;
     NewThread(){//Constructor
      t= new Thread (this, "Demo Thread");      System.out.println("Child thread: "+t);    t.start();
     }//NewThread Constructor
     NewThread(String ThreadName){ // Constructor overloading
      Name=ThreadName;      t= new Thread (this, Name);
      System.out.println("New thread: "+t);      t.start();
     }//NewThread (ThreaName) Constructor
      public void run(){
      try{
            for (int n=5; n>0; n--){
                System.out.println(Name+"Child Thread"+n);           Thread.sleep(500);
            }//for llop
        } catch (InterruptedException e){
            System.out.println(Name+"Child thread  Interrupted:    ");
                  }//catch
     System.out.println(Name+"Exising Child thread  :    ");
      }//run()
      }//Newthread class
```

Creation of thread: By implementing Runnable interface

```
public class thread {
   public static void main(String[] args) {
    /**  Thread t= Thread.currentThread();
      System.out.println("Current thread: "+t);
      t.setName("My Thread");
      System.out.println("Current thread: "+t);
      */
     new NewThread("One");
     new NewThread("Two");
     new NewThread("Three");
      try{
          for (int n=5; n>0; n--){
              System.out.println("Main Thread: "+n);
              Thread.sleep(15000);
          }
      } catch (InterruptedException e){
          System.out.println("Main thread  Interrupted:    ");
                }//catch
      System.out.println("Main thread  Exiting:    ");
          }//Main()
      }//class thread
```

Creation of thread: By implementing Runnable interface

```
--- exec-maven-plugin:3.0.0:exec
(default-cli) @ RSY ---
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Main Thread: 5
TwoChild Thread5
OneChild Thread5
ThreeChild Thread5
ThreeChild Thread4
OneChild Thread4
TwoChild Thread4
OneChild Thread3
ThreeChild Thread3
TwoChild Thread3
OneChild Thread2
ThreeChild Thread2
TwoChild Thread2
ThreeChild Thread1
TwoChild Thread1
OneChild Thread1
TwoExising Child thread  :
OneExising Child thread  :
ThreeExising Child thread  :
Main Thread: 4
```

Creation of Threads :Inheritance

- The ***run()*** method is where the action of a thread takes place.
- The execution of a thread starts by calling its ***start()*** method.

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime; }
    public void run() {
// compute primes larger than minPrime  . . .
    }
}
```

- The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

Creation of Threads :Inheritance

```
import java.util.Scanner;
//Creating thread implementing Runnable interface
class NewThread extends Thread{
      NewThread(){
      super("Demo Thread");
      System.out.println("Child thread: "+this);
      start();
     }//NewThread Constructor
      //Entrypoint of 2nd thread through overriding the run()
      public void run(){
      try{
            for (int n=5; n>0; n--){
               System.out.println("Child Thread"+n);
               Thread.sleep(500);
            }//for llop
        } catch (InterruptedException e){
            System.out.println("Child thread  Interrupted:    ");
                  }//catch
     System.out.println("Exising Child thread  :    ");
      }//run()
      }//Newthread class
```

Creation of Threads :Inheritance

```
public class CreateThreadExtend {
    public static void main(String[] args) {
        new NewThread();
        try{
            for (int n=5; n>0; n--){
                System.out.println("Main Thread: "+n);
                Thread.sleep(1000);//sleep 1000 milisecond
            }//for loop
        } catch (InterruptedException e){
            System.out.println("Main thread  Interrupted:    ");
                    }//catch
        System.out.println("Main thread  Exiting:    ");
            }//Main()
        }//class CreateThreadExtend
```

Creation of Threads :Inheritance

Output :

```
--- exec-maven-plugin:3.0.0:exec
(default-cli) @ RSY ---
Child thread: Thread[Demo
Thread,5,main]
Main Thread: 5
nullChild Thread5
nullChild Thread4
Main Thread: 4
nullChild Thread3
nullChild Thread2
Main Thread: 3
nullChild Thread1
nullExising Child thread  :
Main Thread: 2
Main Thread: 1
Main thread  Exiting:
```

Sleep, Yield, Notify & Wait Thread's Functions

- ***sleep(long millis)*** - causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- ***yield()*** - causes the currently executing thread object to temporarily pause and allow other threads to execute.
- ***wait()*** - causes current thread to wait for a condition to occur (*another thread invokes the* ***notify()*** *method or the* ***notifyAll()*** *method for this object*). This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notify()*** - notifies a thread that is waiting for a condition that the condition has occurred. This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notifyAll()*** – like the ***notify()*** method, but notifies all the threads that are waiting for a condition that the condition has occurred.