Multi-Threading

Thread- The smallest until of dispatchable code

What are java threads?

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process).

A thread is a

- Facility to allow multiple activities within a single process
- Referred as lightweight process
- A thread is a series of executed statements
- Each thread has its own program counter, stack and local variables
- A thread is a nested sequence of method calls
- Its shares memory, files and per-process state

Thread in Java

In Java, the word thread means two different things.

1. An instance of Thread class.
2. or, A thread of execution.

Why multi-threading ?

- Multithreading reduces the CPU idle time that increase overall performance of the system.
- Thread is lightweight process hence it takes less memory and perform context switching
- It helps to share the memory and reduce time of switching between threads.

What about multi-tasking?

- Multitasking is a process of performing multiple tasks simultaneously. We can understand it by computer system that perform multiple tasks like: writing data to a file, playing music, downloading file from remote server at the same time.
- Multitasking can be achieved either by using multiprocessing or multithreading.
- Multitasking by using multiprocessing involves multiple processes to execute multiple tasks simultaneously whereas Multithreading involves multiple threads to executes multiple tasks.

Why Multi-threading ?

- Thread has many advantages over the process to perform multitasking.
- Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system.
- To overcome these issue process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently.
- Hence, computer systems prefer to use thread over the process and use multithreading to perform multitasking.

The main thread

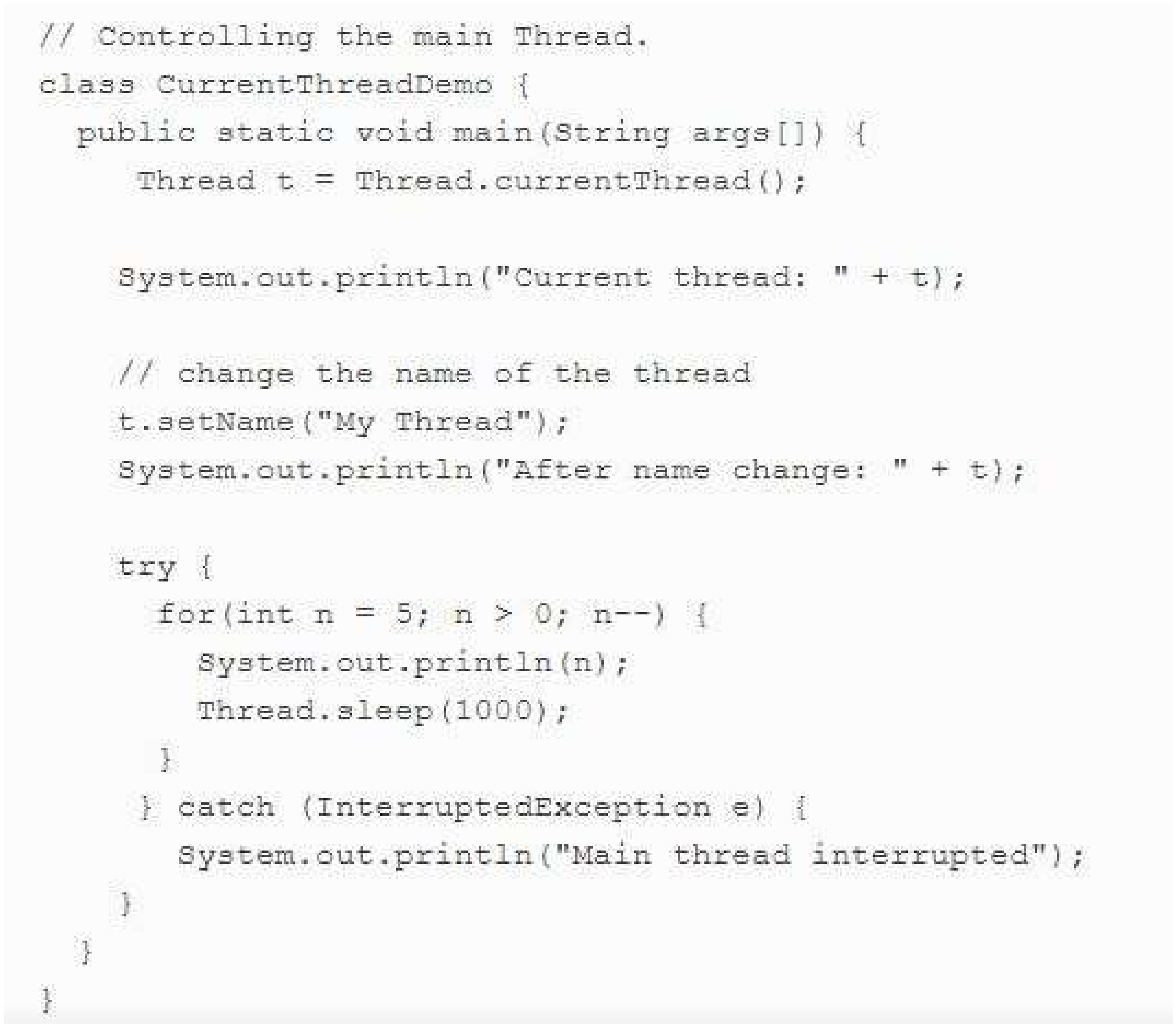
When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

1. It is the thread from which other “child” threads will be spawned.
2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

main( ) can be controlled through a Thread object. For that obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

```
static Thread currentThread( )
```

Code showing how to control main thread



```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
     Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
     } catch (InterruptedException e) {
       System.out.println("Main thread interrupted");
    }
  }
}
```

```
Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5

4

3

2

1
```

sleep( ) setName( ) getName( )

static void sleep(long milliseconds) throws InterruptedException

- The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException.
- The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

final void setName(String threadName)

final String getName( )

- Can set the name of a thread by using setName( ). You can obtain the name of a thread by calling getName( ) (but note that this is not shown in the program). These methods are members of the Thread class.
- In the setName( ) threadName specifies the name of the thread.

Creating threads

Thread implementation in java can be achieved in two ways:

1. Extending the java.lang.Thread class
2. Implementing the java.lang.Runnable Interface

Thread creation

Implementing Runnable

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- Runnable abstracts a unit of executable code.
- Can construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run( ).

public void run( )

- Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.
- This thread will end when run( ) returns.

Implementing Runnable

- Create a class that implements the Runnable interface
- Instantiate an object of type Thread from within that class
- After the new thread is created, call its start( ) method, which is declared within Thread. start( ) executes a call to run( ).

Thread defines several constructors.

Thread(Runnable threadOb, String threadName)

- threadOb is an instance of a class that implements the Runnable interface
- The name of the new thread is specified by threadName.

```java
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
  } catch (InterruptedException e) {
    System.out.println("Child interrupted.");
  }
  System.out.println("Exiting child thread.");
 }
}
```

```java
class ThreadDemo {
  public static void main(String args[ ] ) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {

      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Thread creation

Extending Thread

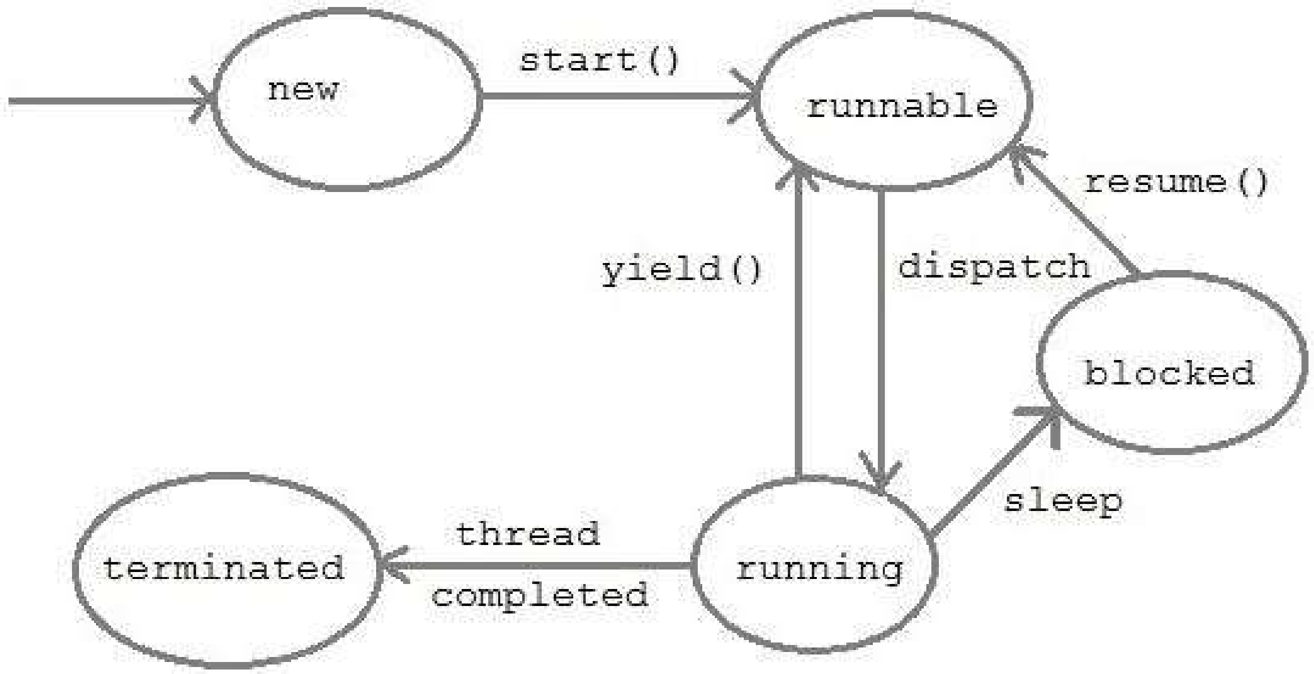
Extending Thread class

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run( ) method, which is the entry point for the new thread.
- It must also call start( ) to begin execution of the new thread.

```java
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```java
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Thread life cycle



Thread life cycle

Thread life-cycle

1. New : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. Runnable : After invocation of start() method on new thread, the thread becomes runnable.
3. Running : A thread is in running state if the thread scheduler has selected it.
4. Waiting : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. Terminated : A thread enter the terminated state when it complete its task.

How can one thread know when another thread has ended?

USING ISALIVE( ) AND JOIN( )

Two ways exist to determine whether a thread has finished.

1. Call isAlive( ) on the thread
2. The method that is commonly use to wait for a thread to finish is called join( )

final boolean isAlive( )

The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

THREAD PRIORITIES

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

Thread Priority

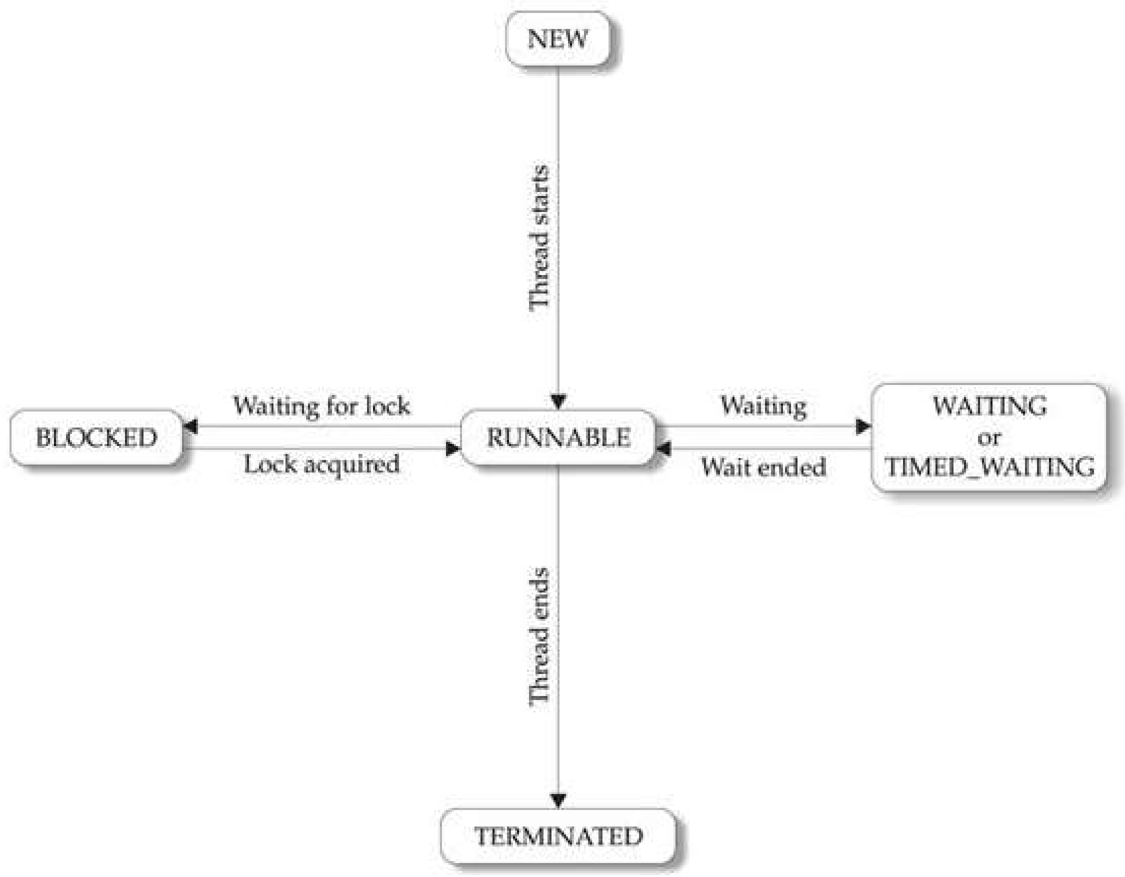
- The amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also preempt a lower-priority one.
- To set a thread's priority, use the setPriority( ) method, which is a member of Thread.

final void setPriority(int level)

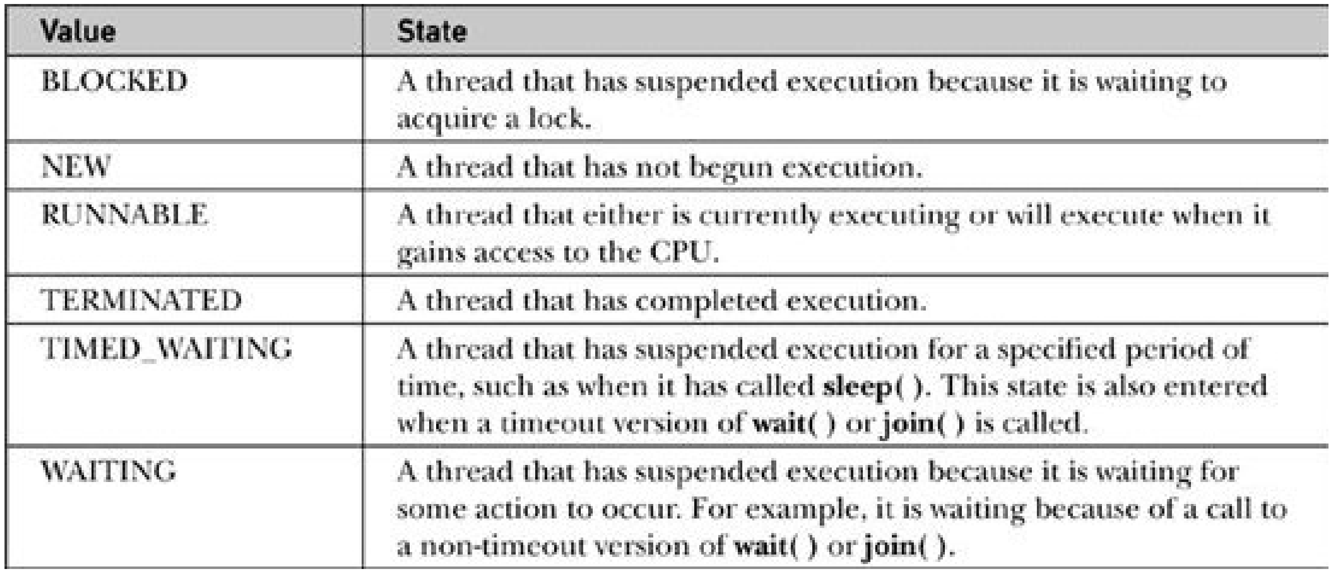
- *level* specifies the new priority setting for the calling thread.
  - The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as static final variables within Thread.

You can obtain the current priority setting by calling the **getPriority**( ) method of Thread.[final int getPriority( )]

Thread states


$$\begin{tikzcd}
 & \text{NEW} \arrow[d, "\text{Thread starts}"] & \\
\text{BLOCKED} \arrow[r, shift right, "\text{Lock acquired}"'] & \text{RUNNABLE} \arrow[l, shift right, "\text{Waiting for lock}"'] \arrow[r, shift left, "\text{Waiting}"] \arrow[d, "\text{Thread ends}"] & \text{WAITING or TIMED\_WAITING} \arrow[l, shift left, "\text{Wait ended}"] \\
 & \text{TERMINATED} &
\end{tikzcd}$$

States of thread



Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.

Given a Thread instance, you can use getState( ) to obtain the state of a thread. For example, the following sequence determines if a thread called thrd is in the RUNNABLE state at the time getState( ) is called:

```
Thread.State ts = thrd.getState();
if(ts == Thread.State.RUNNABLE) // …
```

Synchronization

Synchronization

Synchronized Method

Synchronized Block

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.

Suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file temporary.txt which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using temporary.txt file, this file will be locked(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

Using Synchronized Methods

- To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

- The objectidentifier is a reference to an object whose lock associates with the monitor that the synchronized statement represents.

```
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}
```

```
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }


  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
  }
}
```

```
class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```