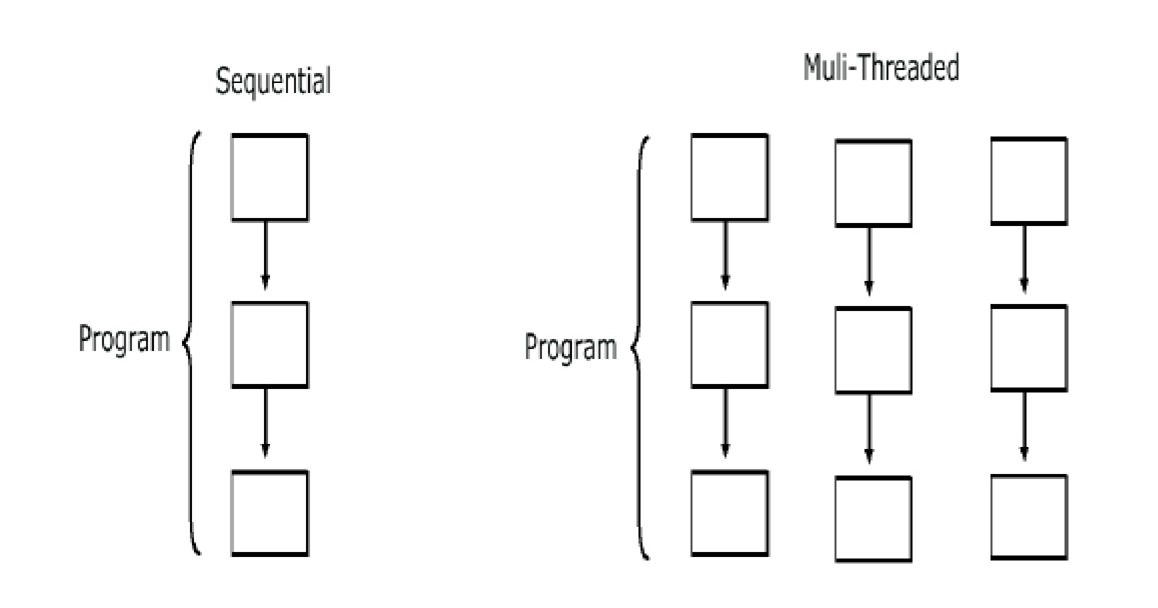
Threads

SUJATA BATRA

Why Threads

- Need to handle concurrent processes
- Definition
  - Single sequential flow of control within a program
- think of threads as processes executed by a program
- Example:
  - Operating System
  - HotJava web browser



Multi-threading in Java Platform

- We start with just one thread, called the main thread. This thread has the ability to create additional threads

- *When a new* thread is started , a new stack materializes and methods called from *that* thread run in a call stack that’s separate from the main() call stack.

- That second new call stack runs concurrently with the main thread

- *A thread of execution* is an individual process (a “lightweight” process) that has its own call stack.
- *one thread per call stack—one call stack per thread*.

The Main Thread

- Every java program has atleast one thread called main thread
- The current thread method of Thread class gets the current thread
- getName() and setName() method are used to get the name and set the name for the thread

```
class setname {
    public static void main(String args[])      {
        Thread thread = Thread.currentThread();
 System.out.println("Main thread's original name is " +
   thread.getName());
        thread.setName("The Main Thread");
        System.out.println("Main thread's name is now " +
            thread.getName());
    }
}
```

Making a Thread

- A thread a begins as an instance of java.lang.Thread.

```
public void run() {
    // your job code goes here
  }
```

- Put the code that needs to be run in a separate thread in a run() method.

- The run() method will call other methods, but the thread of execution of the new call stack always begins by invoking run()

Instantiate a thread

- Extend the java.lang.Thread class
  - Extend Thread when you have a more specialized version of a thread class.
  - The limitation with this approach is that if you extend Thread, *you can’t extend anything else*.
- Implement the Runnable interface
  - design a class that implements the Runnable interface, leaves your class free to extend from some *other* class.

Extend the java.lang.Thread class

- To define code to run in a separate thread
  - Extend the Thread class.
  - Override the run() method.

```
    class MyThread extends Thread {
    public void run() {
    System.out.println(“Starting My Thread");
    }
    }
public class Example_1 {
   public static void main (String args[]) {
    MyThread mt1=new MyThread();
    mt1.start();
    }
}
```

Implementing java.lang.Runnable

- Implementing the Runnable interface gives a way to extend from any class but still define behavior that will be run by a separate thread.

```
class ExRunnable implements Runnable {
public void run()  {
System.out.println("Important job running in MyRunnable");
}
}
public class Example_3 {
public static void main (String [] args) {
ExRunnable r = new ExRunnable();
Thread t = new Thread(r);
t.start();
}
```

- Instantiate Runnable class and get an instance of thread - *give it the job*
- Thread class for the *thread-specific* code
- Thread objects is created by calling the Thread constructor that takes a Runnable argument. The Runnable object is the *target* of the thread.

Starting a Thread

- To launch a new call stack.
  - t.start();
- Prior to calling start() -it is in the *new* state as Thread *object*
  - A new thread of execution starts (with a new call stack).
  - The thread moves from the *new* state to the *runnable* state.
  - When the thread gets a chance to execute, its target run() method will run.

`void start()`

  - Creates a new thread and makes it runnable

`void run()`

  - The new thread begins its life inside this method

Starting Multiple Thread

```
class NameRunnable implements Runnable {
public void run() {
for (int x = 1; x < 4; x++) {
System.out.println("Run by " + Thread.currentThread().getName());
   }
  }
 }
public class Example_5 {
public static void main (String [] args) {
NameRunnable nr = new NameRunnable(); // Make one Runnable
Thread one = new Thread(nr);
one.setName("ash");
Thread two = new Thread(nr);
two.setName("bash");
Thread three = new Thread(nr);
three.setName("clash");
one.start();
two.start();
three.start();
}
}
```

The Thread Scheduler

- The thread scheduler is the part of the JVM
- Any thread in the *runnable* state can be chosen by the scheduler to be the one and only *running* thread.
- **java.lang.Thread Class**
  - public static void sleep(long millis) throws InterruptedException
  - public static void yield()
  - public final void join()
  - public final void setPriority(int newPriority)
- **java.lang.Object Class**
  - public final void wait()
  - public final void notify()
  - public final void notifyAll()

Thread States

- **New**
  - When the Thread instance has been instantiated, but the start() method has not been invoked on the thread.
  - It is a live Thread object, but not yet a thread of execution.
  - At this point, thethread is considered *not alive*.
- **Runnable**
  - The state a thread is in when it's eligible to run,
  - Scheduler has not selected it to be the running thread.
  - A thread first enters the runnable state when the start() method is invoked
  - A thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state.
  - When the thread is in the runnable state, it is considered *alive*.

SUJATA BATRA

Thread States

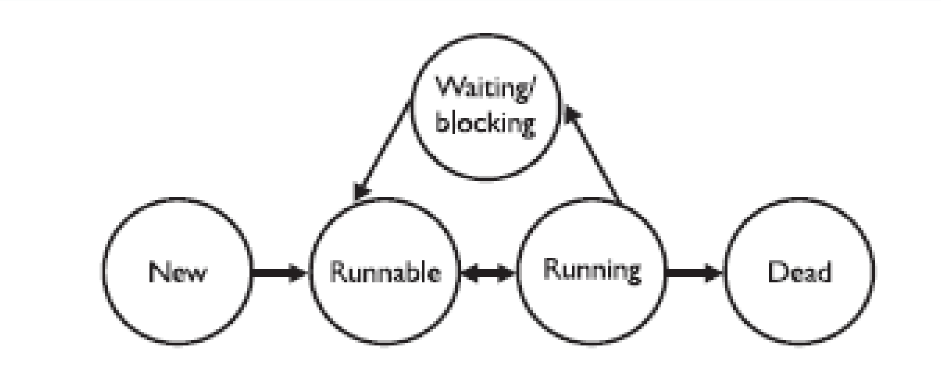
- **Running**
  - This is the state a thread is in when the thread scheduler selects it from the runnable pool to be the currently executing process.
  - A thread can transition out of a running state for several reasons, including because “the thread scheduler felt like it.”

- **Waiting/blocked/sleeping**
  - The thread is still alive, but is currently not eligible to run.
  - A thread may be *blocked* waiting for a resource
  - A thread may be *sleeping* because the thread’s run code *tells* it to sleep for some period of time
  - *waiting*, because the thread’s run code *causes* it to wait

SUJATABATRA

Thread States

- **Dead**
- A thread is considered dead when its run() method completes.
- It may still be a viable Thread object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life
- A runtime exception will be thrown



Non Daemon Vs Daemon Thread

Non Daemon Thread/ User Threads	Daemon Thread
JVM wait until user threads to finish their work. It never exit until all user threads finish their work.	The JVM will’t wait for daemon threads to finish their work. The JVM will exit as soon as all user threads finish their work.
JVM will not force to user threads for terminating, so JVM will wait for user threads to terminate themselves.	If all user threads have finished their work JVM will force the daemon threads to terminate
User threads are created by the application.	Mostly Daemon threads created by the JVM.
Mainly user threads are designed to do some specific task.	Daemon threads are design as to support the user threads.
User threads are foreground threads.	Daemon threads are background threads.
User threads are high priority threads.	Daemon threads are low priority threads.
Its life independent.	Its life depends on user threads.

Leave the running state

- **sleep()** is Guaranteed to cause the current thread to stop executing for at least the specified sleep duration (although it might be *interrupted* before its specified time).

- **yield()** Not guaranteed to do much of anything, although typically it will cause the currently running thread to move back to runnable so that a thread of the same priority can have a chance.

- **join()** Guaranteed to cause the current thread to stop executing until the thread it joins with the thread it calls wait on completes.

- The thread might leave the running state in the following scenarios
  - The thread's run() method completes..
  - A thread can't acquire the *lock* on the object whose method code it's attempting to run

The sleep() method

- The sleep() method is a static method of class Thread.
- “slows a thread down” by forcing it to go into a sleep mode before coming back to runnable

```
     try {
 Thread.sleep(5*60*1000); // Sleep for 5 minutes
 } catch (InterruptedException ex) { }
```

- sleep() code can be put anywhere,
- When the executing code hits a sleep() call, it puts the currently running thread to sleep.
- When a thread wakes up it simply goes back to the runnable state.
- Time specified in sleep() is the minimum duration in which the thread won’t run, but it is not the exact duration in which the thread won’t run.

Pausing a Thread-sleep()

- Sleep Method is used to pause the thread for a amount of time in milliseconds
- The thread will wait for that amount of time before continuing

```
class Example_4
{
    public static void main(String args[])
    {
        try {
            System.out.println("Hello");
            Thread.sleep(1000);
            System.out.println("from");
            Thread.sleep(1000);
            System.out.println("Java.");
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}
```

The Join() Method

- The non-static join() method of class Thread lets one thread "join onto the end" of another thread.

- The thread class join method waits until a thread is finished executing or waiting for a thread to die before returning .

Joining Thread

```
public static void main(String[] args) {

       Thread t = new Thread( ) {

            public void run( ) {

                System.out.println("Reading");

                try {

                    System.in.read( );

                } catch (java.io.IOException ex) {

                   }

                System.out.println("Thread Finished.");
            }
        };
```

Joining Thread

```
        System.out.println("Starting");

        t.start( );

        System.out.println("Joining");

        try {

            t.join( );

        } catch (InterruptedException ex) {

             ex.printStackTrace();

        }

        System.out.println("Main Finished.");

    }
```

Thread Priorities

- Threads always run with some priority, represented usually as a number between 1 and 10
- The scheduled of JVMs use thread priorities in one important way:
- The lower-priority running thread usually will be bumped back to runnable and the highest-priority thread will be chosen to run.
- The running thread will be of equal or greater priority than the highest priority threads in the pool.
- The Thread class has three constants (static final variables) that define the range of thread priorities:
  - Thread.MIN_PRIORITY (1)
  - Thread.NORM_PRIORITY (5)
  - Thread.MAX_PRIORITY (10)

Setting a Thread’s Priority

A thread gets a default priority that is the priority of the thread of execution that creates it.

```
public class TestThreads {
public static void main (String [] args) {
MyThread t = new MyThread();
}
}
```

the thread referenced by *t* will have the same priority as the *main* thread,

since the main thread is executing the code that creates the MyThread instance.

set a thread’s priority directly by calling the setPriority() method on a Thread instance as follows:

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8);
t.start();
```

Thread.yield()

- yield() is *supposed* to make the currently running thread head back to runnable to allow other threads of the *same* priority to get their turn.
- yield() promotes graceful turn-taking among equal-priority threads.
- yield() method isn't guaranteed to do what it claims, and even if yield() *does* cause a thread to step out of running and back to runnable, *there's no guarantee the yielding thread won't just be chosen again over all the others!*
- yield() might—and often does—make a running thread give up its slot to another runnable thread of the same priority, there's no guarantee.