# Threads
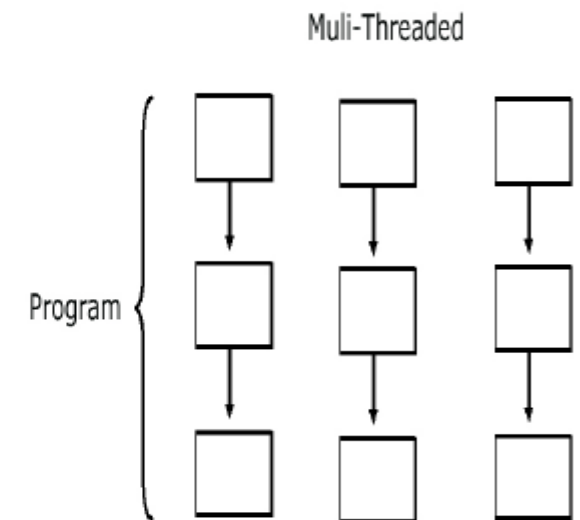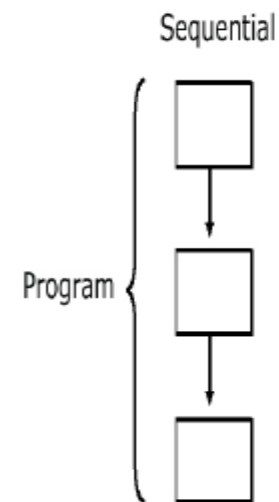
Sujata Batra

# 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

Sequential

Program {

Muli-Threaded

Program {

## 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()

- 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.

- 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.

```java
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.

▶ 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

```java
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()

- **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*.

- **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

▶ **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
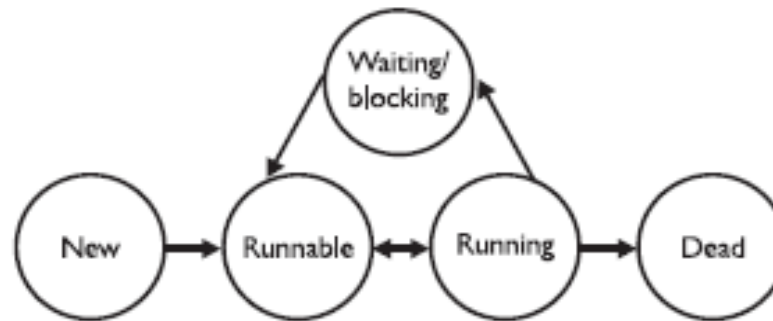
## Leave the running state

▶ **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

```java
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

```java
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.

```java
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:

```java
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.

# RACE CONDITION

▶ A race condition is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled. Race conditions can lead to unpredictable results and subtle program bugs. A thread can prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked.

## Synchronizing Code

- Threads can share resources.

- Sometimes it is desirable that only one thread at a time has access to a shared resource.

- Achieved by using the keyword: `synchronized`, Only *methods* can be synchronized, not variables.

- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods

- A lock can be associated with a shared resource. Threads gain access to a shared resource by first acquiring the lock associated with the resource.

- one thread can hold the lock and thereby have access to the shared resource.

## Synchronizing Code

▶ A class can have both synchronized and no synchronized methods.

▶ If a class has both synchronized and nonsynchronized methods, *multiple threads can still access the nonsynchronized methods* of the class!

▶ *A thread can acquire more than one lock.*

▶ A thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring *that* lock as well.

▶ As the stack unwinds, locks are released again.

## Thread Synchronization

```java
class TwoStrings {

synchronized static void print(String str1, String str2) {

System.out.print(str1);

try {
Thread.sleep(500);
} catch (InterruptedException ie)
{
ie.printStackTrace();
}
System.out.println(str2);

}

}
```

# Thread Synchronization

```
class PrintStringsThread implements Runnable {

String str1, str2;

PrintStringsThread(String str1, String str2) {

    this.str1 = str1;
    this.str2 = str2;

 Thread thread = new Thread(this);
 thread.start();
}


public void run() {
 TwoStrings.print(str1, str2);

 }
}
```

# Thread Synchronization

```java
public class Test {

public static void main(String args[])

{

new PrintStringsThread("Hello ", "there.");

 new PrintStringsThread("How are ", "you?");

 new PrintStringsThread("Thank you ","very much!");

 new PrintStringsThread("Bye ","Bye!");


}

}
```

**Output- May be in Different Order**

**Hello there**

**How are you**

**Thank you very much**

**Bye,Bye**

# Thread Interaction

- The java.lang.Object class has three methods—wait(), notify(), and notifyAll()—that help threads communicate

- wait(), notify(), *and* notifyAll() *must be called from within a synchronized context!*

- If many threads are waiting on the same object, only one will be chosen (in no guaranteed order) to proceed with its execution. If there are no threads waiting, then no particular action is take

# Difference between Blocked-Waiting

- **Blocked**
- Thread is waiting to get a lock on the monitor.(or waiting for a blocking i/o method)
- Caused by the thread tried to execute some synchronized code. (or a blocking i/o method)
- Can move to ready only when the lock is available. ( or the i/o operation is complete)

- **wait**
- Thread has been asked to wait. (by means of wait method)
- The thread already acquired the lock and executed some synchronized code before coming across a wait call.
- Can move to ready only when it gets notified (by means of notify or notifyAll)

## notifyAll()

▶ This notifies all waiting threads and they start competing to get the lock. As the lock is used and released by each thread, all of them will get into action without a need for further notification.

▶ Object can have many threads waiting on it, and using notify() will affect only one of them.

▶ Which one exactly is not specified and depends on the JVM implementation,

▶ never rely on a particular thread In cases in which there might be a lot more waiting, the best way to do this is by using notifyAll().

# Notify -Example

```
class Computations extends Thread
{
    private int num;
    private boolean isComplete;
    private int result;

    public Computations(int num)  {
        this.num = num;
    }
    public synchronized void run()    {
        result = num * 2;
        isComplete=true;
        notify();
    }
```

# Notify -Example

```
public synchronized int getResult() {
    while(!isComplete)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
    }
    return result;
}

}
```

# Notify -Example

```
public class ThrNotifyWait {

    public static void main(String[] args) {

        Computations[] comp = new Computations[4];

        for(int i =0;i<4 ;i++)
        {
        comp[i] = new Computations(i);
        comp[i].start();
        }


        for(Computations c :comp)
            System.out.println(c.getResult());

    }

}
```

# Concurrency

▶ Java platform always has a strong support for concurrent programming and multithreading.

▶ earlier days the support was in the form of calling native constructs itself in the application layer.

▶ Before java 1.5, multithreading applications were created using thread group, thread pool or custom thread pool.

▶ Entire thread management was the responsibility of the programmer keeping in mind the following points.

   ▶ Thread synchronization

   ▶ Thread waiting

   ▶ Thread joining

   ▶ Thread locking

   ▶ Thread notification

   ▶ Handling dead lock

▶ Disadvantage

   ▶ Handle those primitive construct calls efficiently. Otherwise the application will not run properly and unexpected results will be generated.

# Thread Pool

- A thread pool is a group of pre-instantiated, idle threads which stand ready to be given work.

- Preferred over instantiating new threads for each task when there is a large number of short tasks to be done rather than a small number of long ones.

-  Prevents having to incur the overhead of creating a thread a large number of times.

# Executor Framework

- Introduced in Java 1.5 .

- Part of java concurrency package.

- Abstraction layer over the actual implementation of java multithreading.

  - hides the critical parts of concurrent execution and the programmer only concentrates on the business logic implementation.

- First concurrent utility framework in java ,used for standardizing invocation, scheduling, execution and control of asynchronous tasks in parallel threads.

- Executor implementation thread pools which consists of worker threads.

# Executor Framework

▶ Parallel works are considered as tasks instead of simple threads.

▶ Application deals simply with instances of Runnable (which is basically collections of tasks or parallel works) and then it is passed to an **Executor** to process.

▶ The **ExecutorService** interface extends the simplistic Executor interface.

▶ The Java Executor framework has life cycle methods to manage the entire concurrent execution flow.

# Executor Framework
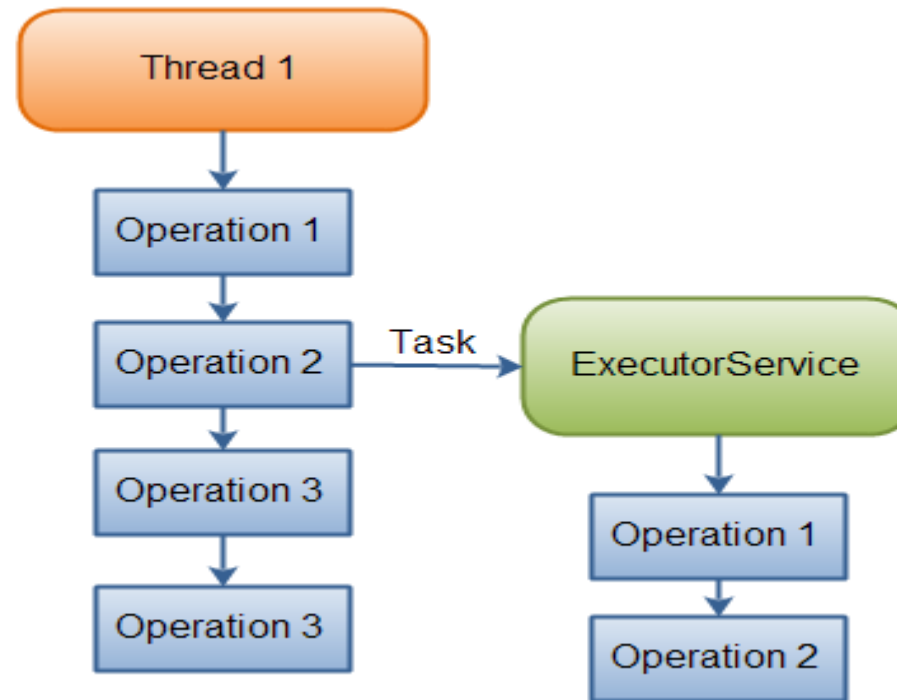
# Creating Thread Pool

This class java.util.concurrent.Executors provides the following factory methods to create thread Pool:

- newFixedThreadPool() creates and executor with the *fixed thread pool*.

- The newCachedThreadPool() creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.

- The newSingleThreadExecutor() creates an executor that executes a single task at a time.

# Executor Interfaces

The java.util.concurrent package defines the following executor interfaces:

▶ Executor, a simple interface that supports launching new tasks.

▶ ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.

# The Executor Interface

▶ Provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom.

▶ If r is a Runnable object, and e is an Executor object you can replace

  (new Thread(r)).start();   with  e.execute(r);

# The ExecutorService Interface

▶ Supplements execute with a similar(), but more versatile submit method.

▶ submit() accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value.

▶ submit() returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

▶ Provides methods for submitting large collections of Callable objects.

   ▶ invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable.

   ▶ The invokeAll() method invokes all of the Callable objects passed as parameter, returns a list of Future objects .

# Shutdown Pool

- ExecutorService provides() a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

  - shutdown() method will not shut down immediately

  - Will no longer accept new tasks.

  - once all threads have finished current tasks, the ExecutorService shuts down.

  - All tasks submitted to the ExecutorService before shutdown() is called, are executed.

  - to shut down the ExecutorService immediately, call shutdownNow() method.

| Modifier and Type | Method and Description |
|---|---|
| boolean | **awaitTermination**(long timeout, **TimeUnit** unit)Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first. |
| <T> **List**<**Future**<T>> | **invokeAll**(**Collection**<? extends **Callable**<T>> tasks)Executes the given tasks, returning a list of Futures holding their status and results when all complete. |
| <T> **List**<**Future**<T>> | **invokeAll**(**Collection**<? extends **Callable**<T>> tasks, long timeout, **TimeUnit** unit)Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first. |
| <T> T | **invokeAny**(**Collection**<? extends **Callable**<T>> tasks)Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do. |
| <T> T | **invokeAny**(**Collection**<? extends **Callable**<T>> tasks, long timeout, **TimeUnit** unit)Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses. |
| boolean | **isShutdown**()Returns true if this executor has been shut down. |
| boolean | **isTerminated**()Returns true if all tasks have completed following shut down. |
| void | **shutdown**()Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. |
| **List**<**Runnable**> | **shutdownNow**()Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. |
| <T> **Future**<T> | **submit**(**Callable**<T> task)Submits a value-returning task for execution and returns a Future representing the pending results of the task. |
| **Future**<?> | **submit**(**Runnable** task)Submits a Runnable task for execution and returns a Future representing that task. |
| <T> **Future**<T> | **submit**(**Runnable** task, T result)Submits a Runnable task for execution and returns a Future representing that task. |

# Callable and Future Interfaces

▶ Callable has just one method call() which holds all the code needs to executed asynchronously.

▶ In Runnable interface, there was no way to return the result of computation or throw checked exception but with Callable you can both return a value and can throw checked exception.

▶ You can use get() method of Future to retrieve result once computation is done. You can check if computation is finished or not by using isDone() method.

▶ You can cancel the computation by using Future.cancel() method.

▶ get() is a blocking call and it blocks until computation is completed.

▶ overloaded get() method with timeout: for definite blocking.