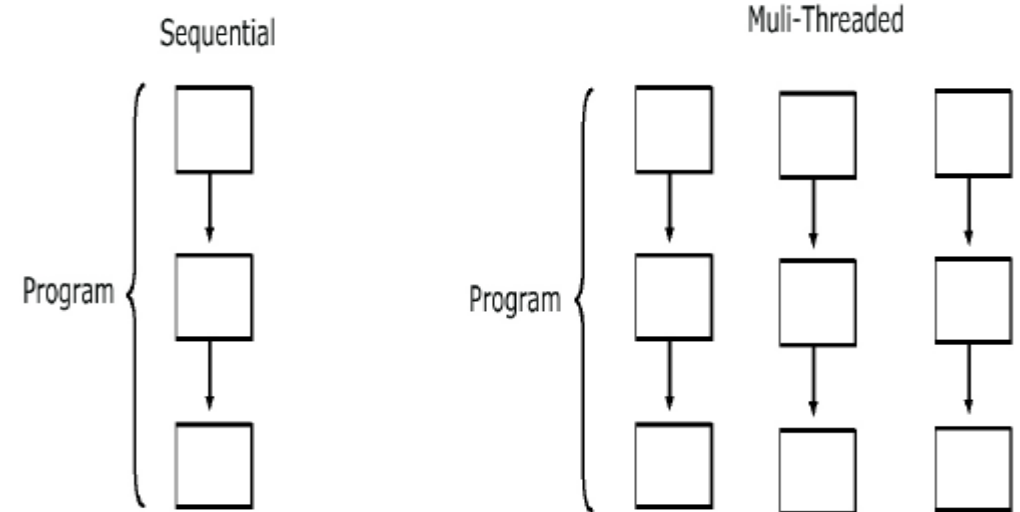


# Threads

# 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 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, the thread 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*.

## 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.”

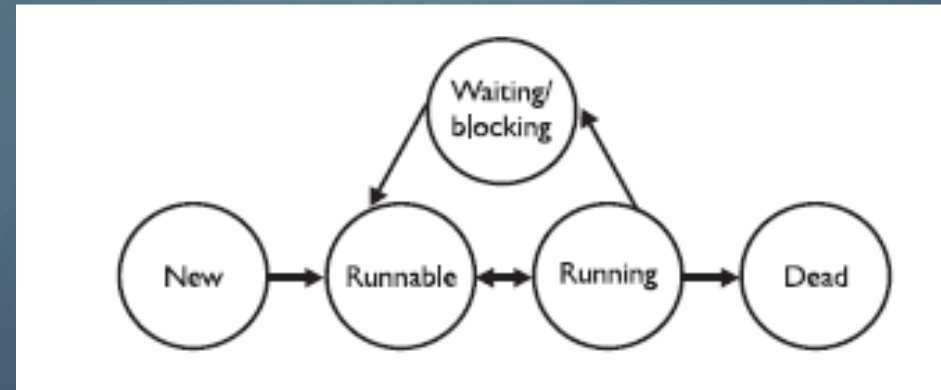
### ▶ **Waiting/blocked/sleeping**

- ▶ 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

## 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.

# 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

```
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

```
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

# Volatile

- ▶ Volatile keyword is used to modify the value of a variable by different threads.
- ▶ Used to make classes thread safe.
- ▶ The volatile keyword does not cache the value of the variable and always read the variable from the main memory.
- ▶ Use the volatile keyword with variables. Using volatile keyword with classes and methods is illegal.

# Example

```
public class VolatileTest {
    private static volatile int MY_INT = 0;
    public static void main(String[] args)
    {
        new ChangeListener().start();
        new ChangeMaker().start();
    }

    static class ChangeListener extends Thread {
        @Override public void run()
        {
            int local_value = MY_INT;
            while (local_value < 5) {
                if (local_value != MY_INT) {
                    System.out.println("Got Change for MY_INT : "+MY_INT);
                    local_value = MY_INT;
                }
            }
        }
    }
}
```

```
static class ChangeMaker extends Thread {
    @Override public void run()
    {
        int local_value = MY_INT;
        while (MY_INT < 5) {
            System.out.println(
                "Incrementing MY_INT to "+(local_value +
1));

            MY_INT = ++local_value;
            try {
                Thread.sleep(500);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

# Volatile Vs Synchronization

Volatile Keyword	Synchronization Keyword
Volatile keyword is a field modifier.	Synchronized keyword modifies code blocks and methods.
The thread cannot be blocked for waiting in case of volatile.	Threads can be blocked for waiting in case of synchronized.
It synchronizes the value of one variable at a time between thread memory and main memory.	It synchronizes the value of all variables between thread memory and main memory.
Volatile fields are not subject to compiler optimization.	Synchronize is subject to compiler optimization.

## 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().



# Producer Consumer Problem

- ▶ In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.
- ▶ The producer's job is to generate data, put it into the buffer, and start again.
- ▶ At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

## Problem

**To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.**

# Producer Consumer Problem

## Solution

- ▶ The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- ▶ An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

# Producer Consumer Example

```
import java.util.LinkedList;

public class Threadexample {
    public static void main(String[] args)
        throws InterruptedException
    {
        // Object of a class that has both produce()
        // and consume() methods
        final PC pc = new PC();

        // Create producer thread
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                try {
                    pc.produce();
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
```

# Producer Consumer Example

```
// Create consumer thread
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run()
    {
        try {
            pc.consume();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
```

# Producer Consumer Example

```
// Start both threads
```

```
    t1.start();
```

```
    t2.start();
```

```
    // t1 finishes before t2
```

```
    t1.join();
```

```
    t2.join();
```

```
}
```

```
// This class has a list, producer (adds items to list
```

```
// and consumer (removes items).
```

# Producer Consumer Example

```
// This class has a list, producer (adds items to list
// and consumer (removes items).
public static class PC {

    // Create a list shared by producer and consumer
    // Size of list is 2.
    LinkedList<Integer> list = new LinkedList<>();
    int capacity = 2;
```



# Producer Consumer Example

```
// Function called by producer thread
public void produce() throws InterruptedException
{
    int value = 0;
    while (true) {
        synchronized (this)
        {
            // producer thread waits while list
            // is full
            while (list.size() == capacity)
                wait();

            System.out.println("Producer produced-"
                               + value);
```

```
// to insert the jobs in the list
        list.add(value++);

        // notifies the consumer thread that
        // now it can start consuming
        notify();

        // makes the working of program easier
        // to understand
        Thread.sleep(1000);
    }
}
```

# Producer Consumer Example

```
// Function called by consumer thread
```

```
public void consume() throws  
InterruptedException
```

```
{
```

```
while (true) {
```

```
    synchronized (this)
```

```
    {
```

```
        // consumer thread waits while list
```

```
        // is empty
```

```
        while (list.size() == 0)
```

```
            wait();
```

```
        // to retrieve the first job in the list
```

```
int val = list.removeFirst();
```

```
System.out.println("Consumer consumed-"  
                    + val);
```

```
// Wake up producer thread  
notify();
```

```
// and sleep  
Thread.sleep(1000);
```

```
}
```

```
}
```

```
}}
```

```
}
```

# 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.

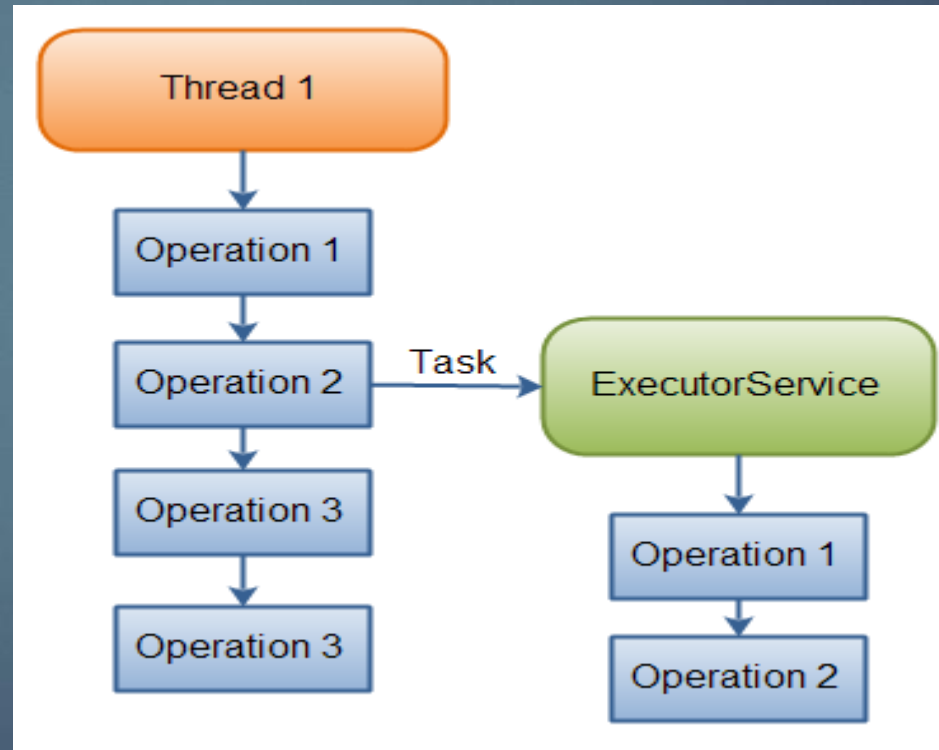
# Executor Framework

- ▶ 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 an 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	<a href="#"><u>awaitTermination(long timeout, TimeUnit unit)</u></a> 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> <a href="#"><u>List Future</u></a> <T>>	<a href="#"><u>invokeAll Collection&lt;? extends Callable&lt;T&gt;&gt; tasks)</u></a> Executes the given tasks, returning a list of Futures holding their status and results when all complete.	
<T> <a href="#"><u>List Future</u></a>	<a href="#"><u>invokeAll Collection&lt;? extends Callable&lt;T&gt;&gt; tasks, long timeout, TimeUnit</u></a>	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	<a href="#"><u>invokeAny(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks)</u></a> Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.	
<T> T	<a href="#"><u>invokeAny(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks, long timeout, TimeUnit unit)</u></a>	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	<a href="#"><u>isShutdown()</u></a> Returns true if this executor has been shut down.	
	<a href="#"><u>isTerminated()</u></a> Returns true if all tasks have completed following shut down.	
Void	<a href="#"><u>shutdown()</u></a> Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.	
<a href="#"><u>List Runnable</u></a>	<a href="#"><u>shutdownNow()</u></a> 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> <a href="#"><u>Future</u></a> <T>	<a href="#"><u>submit Callable&lt;T&gt; task)</u></a> Submits a value-returning task for execution and returns a Future representing the pending results of the task.	
<a href="#"><u>Future</u></a>	<a href="#"><u>submit Runnable task)</u></a> Submits a Runnable task for execution and returns a Future representing that task.	
<T> <a href="#"><u>Future</u></a> <T>	<a href="#"><u>submit Runnable task, T result)</u></a> 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 be 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.

# CountDownLatch

- ▶ CountDownLatch is used to make sure that a task waits for other threads before it starts.



# How CountdownLatch works?

- ▶ When we create an object of CountdownLatch, we specify the number of threads it should wait for, all such thread are required to do count down by calling CountdownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

# Example

```
import java.util.concurrent.CountDownLatch;
```

```
public class CountDownLatchDemo
```

```
{
    public static void main(String args[])
        throws InterruptedException
    {
        // Let us create task that is going to
        // wait for four threads before it starts
        CountDownLatch latch = new CountDownLatch(4);

        // Let us create four worker
        // threads and start them.
        Worker first = new Worker(1000, latch,
                                   "WORKER-1");
        Worker second = new Worker(2000, latch,
                                    "WORKER-2");
```

```
        Worker third = new Worker(3000, latch,
                                   "WORKER-3");
```

```
        Worker fourth = new Worker(4000, latch,
                                    "WORKER-4");
```

```
        first.start();
```

```
        second.start();
```

```
        third.start();
```

```
        fourth.start();
```

```
        // The main task waits for four threads
        latch.await();
```

```
        // Main thread has started
```

```
        System.out.println(Thread.currentThread().getName() +
                               " has finished");
```

```
    }
}
```

# Example

```
class Worker extends Thread
{
    private int delay;
    private CountDownLatch latch;

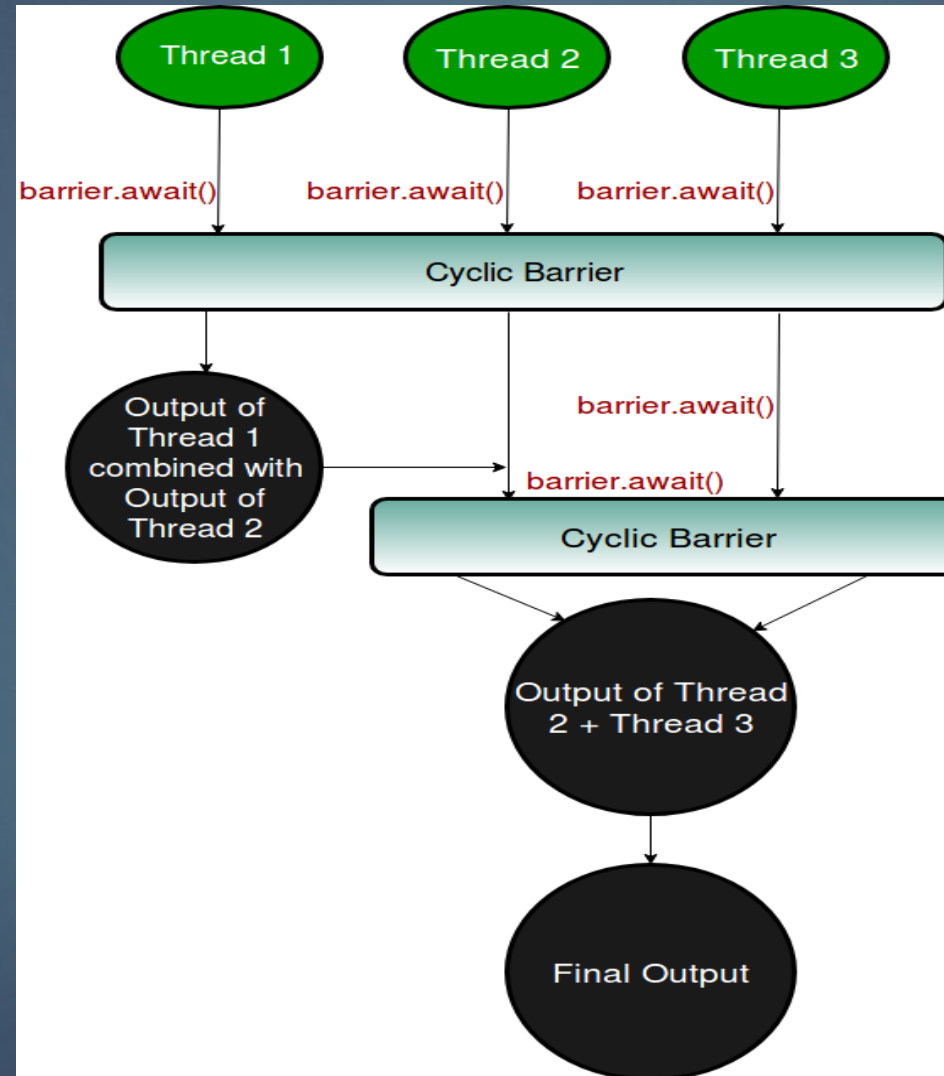
    public Worker(int delay, CountDownLatch
latch,
                String name)
    {
        super(name);
        this.delay = delay;
        this.latch = latch;
    }
}
```

```
@Override
    public void run()
    {
        try
        {
            Thread.sleep(delay);
            latch.countDown();
            System.out.println(Thread.currentThread().getName()
                                + " finished");
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

# CyclicBarrier

- ▶ The cyclic barrier is a concurrent utility mechanism of synchronization that allows a set of threads to all wait for each other to reach a common barrier point.
- ▶ It is useful in Java programs that involve a fixed-sized party of threads that must wait for each other. The word cyclic is used for its reused purpose.
- ▶ It means that the barrier can be re-used when all waiting threads are released. The point is the barrier that ensures all the threads complete their execution before reaching here.
- ▶ Therefore, the mechanism is known as the cyclic barrier.
- ▶ In short, cyclic barrier preserves a count of threads.

# Working of CyclicBarrier



# Methods of CyclicBarrier Class

Method	Description
<code>await()</code>	It returns the arrival index of the current thread.
<code>await(long timeout, TimeUnit unit)</code>	Waits until all parties (threads) have invoked <code>await</code> on this barrier, or the specified waiting time elapses.
<code>getNumberWaiting()</code>	The method is used to return the number of parties currently waiting at the barrier (blocked in <code>await()</code> method). The method is widely used for debugging and assertions.
<code>getParties()</code>	The method returns an integer value that denotes the number of parties required to trip this barrier.
<code>isBroken()</code>	The method returns a boolean value if one or more parties broke out of this barrier due to interruption or timeout since construction or the last reset or a barrier action failed due to an exception. Else, returns false.
<code>reset()</code>	The method is used to reset the barrier to its initial state. The method throws <code>BrokenBarrierException</code> if any parties are currently waiting at the barrier.