Core Java

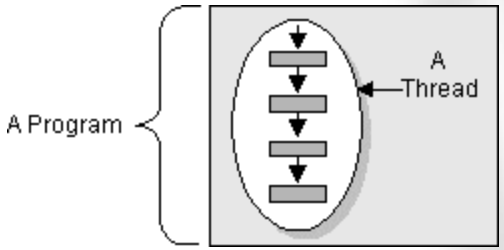
Multithreading in Java

Lesson Objectives

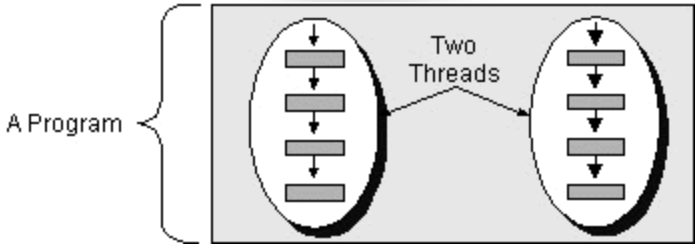
- To understand the following topics
  - Multithreading
  - Main Thread
  - Creating Threads
  - Life Cycle of Thread
  - Scheduling and Priority
  - Concurrency Issues
  - Multithreading Best Practices

What is Multithreading?

- Single threaded program – Single line of execution
  - They have a beginning, sequence and an end at any given time



- Multithreaded application – Multiple line of execution
  - Each have a beginning, sequence and an end of its own
  - They can run in parallel



Advantages of Multithreading

- Maintaining user interface responsiveness
- Waiting for a wake-up call
- Simple Multitasking
- Building multi-user applications
- Multi-processing

Main Thread

- When a Java program starts up, main thread runs.
  - Child threads are spawned from this main thread.
- Last thread to finish execution.
  - When the main thread stops, your program terminates
  - If the main thread finishes before a child thread the Java run-time system may hang.

Methods in Thread Class

Method	Meaning
getName / setName	Obtain / set a thread's name
getPriority	Obtains a thread's priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate
resume	Resume the execution of a suspended thread
run	Entry point for a thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method
suspend	Suspend a thread
currentThread	Returns a reference to the currently executing thread object.
interrupt	Interrupts this thread
yield	Causes the currently executing thread object to temporarily pause and allow other threads to execute

Creating a Thread

- Two ways to create a thread:
  - Extend Thread class
  - Implement Runnable interface

Extend Thread Class to Create Threads

- Class should extend Thread class.
- Inherited class should:
  - Override the *run()* method.
  - Invoke the *start()* method to start the thread.

Implement Runnable Interface

- Easiest way to create a thread.
  - Create a class that implements the runnable interface.
  - Class to implement only the run method that constitutes the new thread.

```
public void run() { . . .}
```

Demo : Creating Threads

Demo:

MainThread.java

Impl.java

Extend.java

Thread Lifecycle

Fig 1:

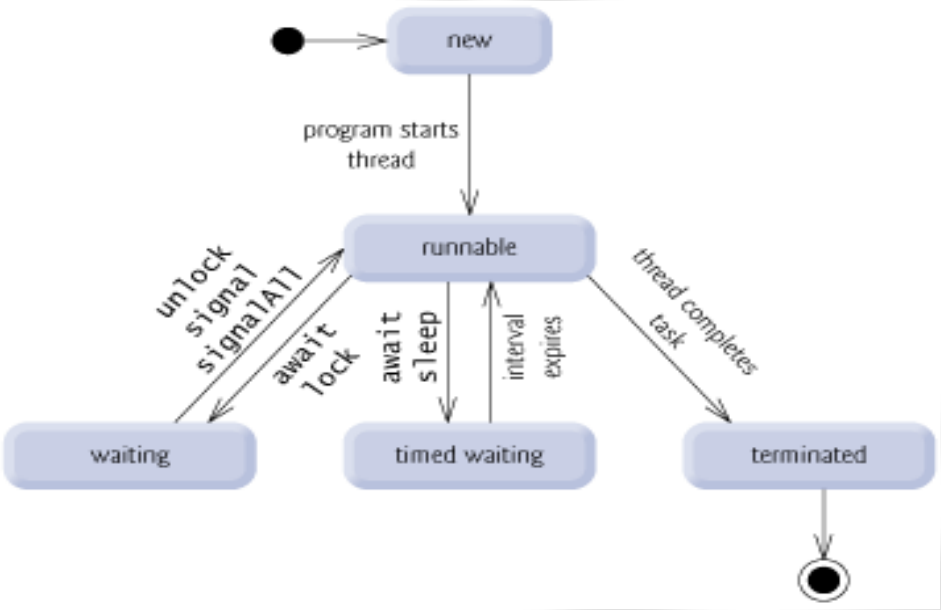
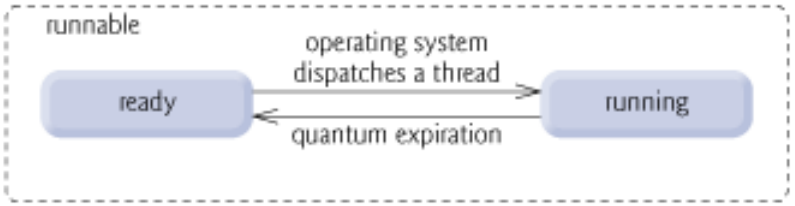


Fig 2:



Scheduling of Threads

- Done by JVM.
- Any thread in the runnable state can be chosen for execution.
- Runnable thread is chosen from a pool and not from a queue.
- Programmer cannot control the scheduler to execute a particular thread.

Priority of Threads

- Threads have priorities ranging from 1 to 10.
- Constants that defines Thread priorities are:
  - Thread.MIN_PRIORITY (1)
  - Thread.NORM_PRIORITY (5)
  - Thread.MAX_PRIORITY (10)
- Default priority is 5.
- Do not rely on thread priorities when you design your multithreaded application.

Priority of Threads (contd..)

- Threads have priorities ranging from 1 to 10.
- Constants that defines Thread priorities are:
  - Thread.MIN_PRIORITY (1)
  - Thread.NORM_PRIORITY (5)
  - Thread.MAX_PRIORITY (10)
- Default priority is 5.
- Do not rely on thread priorities when you design your multithreaded application.

Preemptive and Non Preemptive Scheduling

- Thread scheduling is the mechanism used to determine how runnable threads are allocated *CPU* time.
- Scheduling can be:
  - Preemptive
    - Thread scheduler pauses a running thread to allow different threads to execute.
  - Non-Preemptive
    - Non-preemptive scheduler relies on the running thread to yield control of the CPU so that other threads can execute.

Preemptive and Non Preemptive Scheduling (contd..)

- Thread scheduling is the mechanism used to determine how runnable threads are allocated *CPU* time.
- Scheduling can be:
  - Preemptive
    - Thread scheduler pauses a running thread to allow different threads to execute.
  - Non-Preemptive
    - Non-preemptive scheduler relies on the running thread to yield control of the CPU so that other threads can execute.

JVM Implementation Dependent

- Multithreading is JVM dependent.
- For example, the early Solaris Java platform runs a thread of a given priority to completion or until a higher priority thread becomes ready.

Multithreading Policies

- In 32-bit Java implementations for Win ‘95 & Win NT, threads are time-sliced.
  - Each thread is given a limited execution time on a processor.
  - When that time expires the thread is made to wait.
  - Other threads of *equal* priority get their chance to use their quantum in a round - robin fashion.
- Thus, a running thread can be pre-empted by a thread of equal priority.
- In Solaris, a running thread can only be pre-empted by a thread of higher priority.

Lab : Multithreading

- Lab 10.1

Thread Synchronization

- Synchronization:
  - Technique to ensure that a shared resource is used by only one thread at a time.
  - Avoids race condition.

```
public class counter {
private int count=0;
public int incr()  {
int n = count;
count =  n+1;
return n;
}
}
```

Thread 1	Thread 2	Value of Count
cnt = counter.incr();	---------	0
n = count; [0]	---------	0
----	cnt = counter.incr()	0
----	n = count; [0]	0
---	count = n + 1 [1]	1
	return n; [1]	1
count = n + 1 [1]	--------	1
return n; [0]	--------	1

→ Indicates shifting of process.

-------- Indicates that thread is idle.

Thread Synchronization

- Only methods can be synchronized.
  - Not variables or classes.

```
public class counter {
private int count=0;
public synchronized  int
incr()   {
int n = count;
count =  n+1;
return n;
} }
```

Thread 1	Thread 2	Value of Count
cnt = counter.incr(); (acquires the monitor)	---------	0
n = count; [0]	---------	0
-----	cnt = counter.incr() (can't acquire the monitor)	0
count = n + 1 [1]	blocked	1
return n; [0] [Releases the monitor]	blocked	1
-----	(acquires the monitor)	1
----	n = count; [1]	1
	count = n + 1 [2]	2
	return n; [1]	2
	[release the monitor]	2

Using Synchronized Blocks

- Synchronization affects concurrency.
- If a method scope is more than needed, reduce it to just a *block*:

```
class SyncTest {
public void doMethod() {
System.out.println("not synchronized");
synchronized(this) {
System.out.println("synchronized"); }
} }
```

Using Synchronized Blocks (contd..)

- Synchronization affects concurrency.
- If a method scope is more than needed, reduce it to just a *block:*

```
class SyncTest {
public void doMethod() {
System.out.println("not synchronized");
synchronized(this) {
System.out.println("synchronized"); }
} }
```

Demo : Using Synchronized Blocks

Demo:

ProducerConsumer.java

Lab : Using Synchronized Blocks

- Lab 10.2

Issues with Old Thread Model

- Code related to thread creation and task delegation to the thread is a part of the application itself.
- What if you have multiple helper tasks or a scenario where every single user action requires a new Thread to be spawned to process?
- Creating a lot many threads with no bounds to the maximum threshold can cause an application to run out of memory.

Issues with Old Thread Model (contd..)

- Although threads are light-weight, lot of resources are utilized to create them.
- In such a situation, having a ThreadPool is a better solution.
  - Only fixed number of Threads are created and re-used later.
- The *run()* method has no way to return any result back to the main thread.

Executors

- Executor framework addresses all the above issues and also provides additional life-cycle management features for the threads.
- Interfaces of Executor framework are:
  - Callable
    - Similar to Runnable interface; has a *call()* method.
    - The only difference is that its call() method returns a value.

Executor Framework

- Executor:
  - **Executor** abstracts Thread creation,
  - Executes all *Runnable* tasks
- ExecutorService:
  - **E**xtends the Executor and is able to execute *Callable* tasks in addition to Runnable tasks.
- ScheduledExecutorService:
  - Allows us to schedule the asynchronous tasks thereby adding support for delayed and periodic task execution.

Thread Pools

- Manage a pool of worker threads.
- Contains a work queue which holds tasks waiting to get executed.
- Can be described as a collection of runnables (work queue) and a connections of running threads.
- Constantly run and check the work query for new work.

Thread Pools (contd..)

- If new work is to be done, they execute this Runnable.
- In case threads return some value (result-bearing threads) then you can use the following:

java.util.concurrent.Callable

An Example: Using Executor Framework

```
import java.util.concurrent.*;
class MyRunnable implements Runnable {
private final long countUntil;
MyRunnable(long countUntil) {   this.countUntil = countUntil; }
public void run() {
long sum = 0;
for (long i = 1; i < countUntil; i++)
{   sum += i;                   }
System.out.println("From Thread:"+sum);
} }
```

An Example: Using Executor Framework (contd..)

```
public class Main {
private static final int NTHREDS = 10;
public static void main(String[] args) {
ExecutorService executor =
    Executors.newFixedThreadPool(NTHREDS);
for (int i = 0; i < 50; i++) {
Runnable worker = new MyRunnable(10 + i);
executor.execute(worker);
}
```

An Example: Using Executor Framework (contd..)

```
// This will make the executor accept no new threads
        // and finish all existing threads in the queue
        executor.shutdown();
        // Wait until all threads are finish
        while (!executor.isTerminated()) {

        }
        System.out.println("Finished all threads");
    }
}
```

Futures and Callables

- Callables:
  - Allows to return values after computation.
  - Uses generic to define the type of object which is returned
  - If you submit a callable to an executor the framework returns a java.util.concurrent.Future.
    - Use this future to check the status of a callable and to retrieve the result from it.
  - On the executor, use the method *submit* to submit a Callable and to get a future
  - Retrieve the result of the future using the *get()* method.

Demo : Futures and Callables

Demo:

CallableFutures.java