# Threads

## Content

- Introduction
- Threads
- Threads States
- Methods
- Locks
- Executers

# Threads

Introduction

## What is Multitasking ?

Running at the same time (concurrently) and performing different tasks

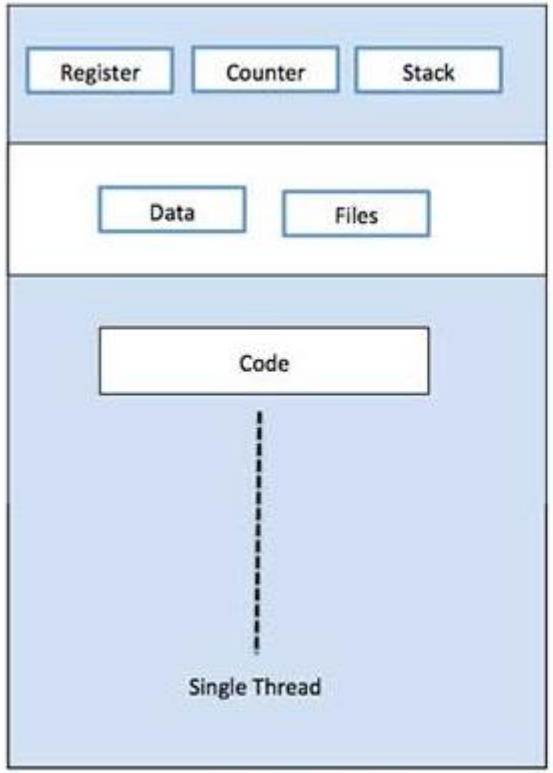
For example, a Web browser can do several things at the same time:

- scroll a page
- download a file
- play animation, sound
- print page
- load a new page

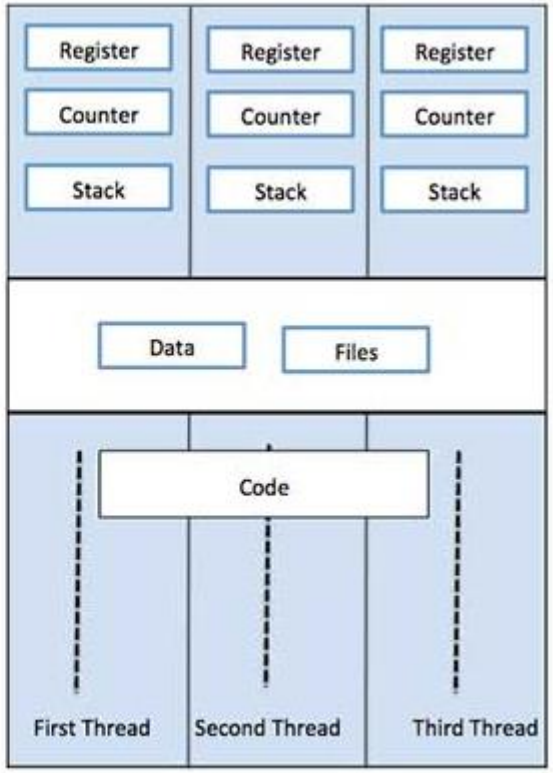
# Threads

Introduction

## What is Multitasking ?




Single Process P with single thread




Single Process P with three threads

# Threads

Introduction

## What is a Thread?

- A Thread is an independent path of execution within a program
- Many threads can run concurrently within a program.
- Every thread in Java is created and controlled by the java.lang.Thread class.
- A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.
- Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority.

# Threads

Introduction

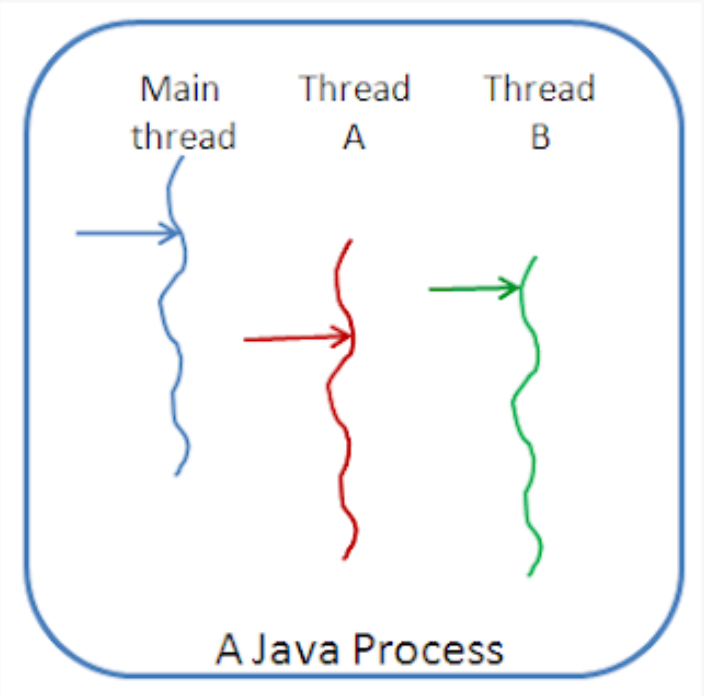
## What is a Process?

- A process has a self-contained execution environment.
- A process generally has a complete, private set of basic run-time resources
- Threads are sometimes called lightweight processes
- Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

# Threads

## Thread VS. Process

Introduction

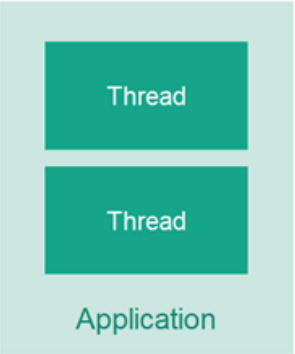


# Threads

## Introduction

## What is a Multithreading ?

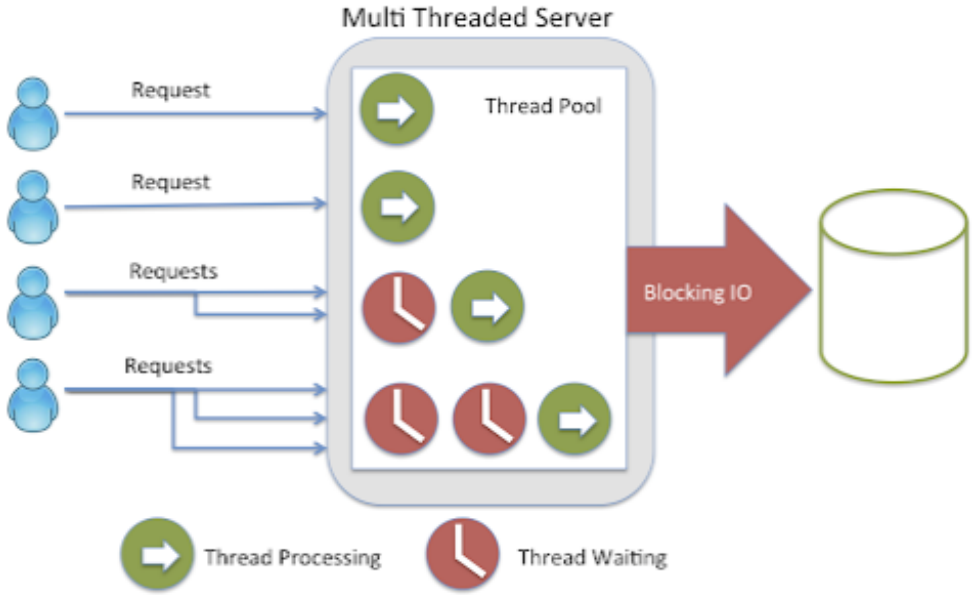
- Multithreading means that you have multiple *threads of execution* inside the same application.
- A thread is like a separate CPU executing your application.
- a multithreaded application is like an application that has multiple CPUs executing different parts of the code at the same time.



# Threads

## Why to use Multithreading ?

Introduction



# Threads

Introduction

## Why to use Multithreading ?

- Improve application performance.
- Better resource utilization.
- Simplify modeling real-world problems.
- Perform asynchronous or background processing.

# Threads

## Create New Thread

There are 2 ways for defining and creating threads

Thread

- Subclassing Thread and Overriding run
- Implementing the Runnable interface

# Threads

## Create New Thread

Thread

**Implementing Runnable**

- Allows your class to extend some other class
- Is good if you want light-weight runnable objects
- That's what interfaces are for
- Problems – might be less convenient for coding

**Sub classing Thread**

- Allows direct access to Thread attributes and methods
- Less coding
- Problems – single inheritance in Java – your class cannot extend another class

Threads

Thread

## Create New Thread

```
public class SimpleThread extends Thread {

//override run and extends thread
public SimpleThread(String str) {
super(str);
}
@Override
public void run() {
for (int i = 0; i < 10; i++) {
        System.out.println(i + " " + getName());
        System.out.println(Thread.currentThread().getPriority());
try {
    sleep((long)(Math.random() * 1000));
    } catch (InterruptedException e) {
        e.getStackTrace();}
        }
    System.out.println("DONE! " + getName()); }

public static void main (String[] args) {
new SimpleThread("thread1").start();
new SimpleThread("thread2").start();

System.out.println("Hi Assaf"
+Thread.currentThread().getPriority());}
}
```

# Threads

Thread

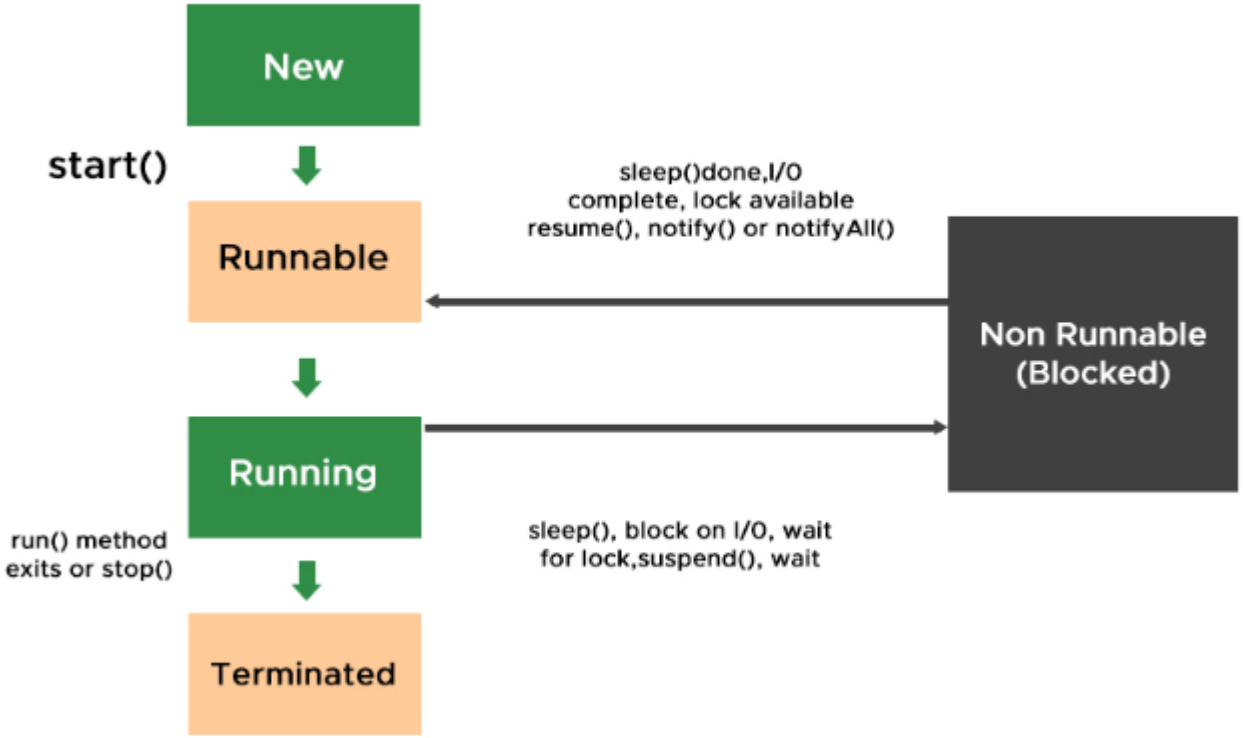
## Create New Thread

```
public class SimpleRunnable implements Runnable {
public void run() {

for (int i = 0; i < 10; i++) {
System.out.println(i + " " + Thread.currentThread().getName());
System.out.println(Thread.currentThread().getPriority());
try {
         Thread.sleep((long)(Math.random() * 3000));
         }
catch (InterruptedException e) {}
                                }
System.out.println("DONE! " + Thread.currentThread(). getName());
}


public static void main (String[] args)
{
 SimpleRunnable  runner=new SimpleRunnable ();
         Thread t1=new Thread(runner, “Apple thread");
         Thread t2=new Thread(runner, “Banana thread");
         t1.start();
         t2.start(); }

}
```

# Threads

## Thread Lifecycle

Thread



# Threads

## Thread States

States

### New Thread

- Thread is created but not yet started.
- No system resource have been allocated for it yet
- Can only be started
- Calling any method besides causes an IllegalThreadStateException

# Threads

States

## Thread States

**Runnable:**

- Thread is ready to run and waiting for **CPU** time.
- When a start() method is called over thread processed by the thread scheduler
    - Case A: Can be a running thread (Running State)
    - Case B: Can not be a running thread (Blocked State)

# Threads

States

## Thread States

### Running

- Creates the system resources necessary to run the thread
- Schedules the thread to run
- Calls the thread's run method

# Threads

States

## Thread States

### Not Runnable (Blocked)

A thread becomes Not Runnable when one of these events occurs:

- It's sleep(), yield() method is invoked.
- One thread uses join() on another and becomes blocked.
- The thread calls the wait() method to wait for a specific condition to be satisfied.
- The thread is blocked on I/O.

# Threads

States

## Thread States

### Terminated (Dead)

- The run method must terminate naturally
- Stop() method – ***deprecated!!!***
    - This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack)

# Threads

Methods

## Thread Methods

- start(): Starts the execution of the thread.
- run(): Entry point for the thread.
- sleep(long millis): Puts the thread to sleep for the specified milliseconds.
- join(): Waits for the thread to die.
- yield(): Causes the currently executing thread to temporarily pause and allow other threads to execute.
- interrupt(): Interrupts the thread.

# Threads

## Start Method

Starting a thread by calling the start() function.

Q: why don't we directly call the overridden run() function?

Methods

A: The purpose of start() is to create a separate call stack for the thread. A separate call stack is created by it, and then run() is called by JVM.

# Threads

Methods

## Sleep Method

- Move the thread to a non-Runnable state for a period (ms)
- Usually, the simplest way to delay threads or main
- Note: blocks the thread at least to the specified time – not exactly
- throws InterruptedException
  - When thread are out of the blocking state before time
  - Might happen due to OS activity

# Threads

Methods

## Join Method

- Move the running thread to a non-Runnable state until a specific thread ends
- Delays the caller until the referenced thread ends
- Is absolute – not like priority
- throws InterruptedException
  - When thread are out of the blocking state before time
  - Might happen due to OS activity

# Threads

Methods

## Yield Method

- Move the Running thread to the Rannable pool (Equals to sleep(0))
- Usually for giving other low priority thread a chance to run
- throws InterruptedException
  - When thread are out of the blocking state before time
  - Might happen due to OS activity

# Threads

Methods

## Suspend Method

- The suspend() method of thread class puts the thread from running to waiting state.
- The suspended thread is often resumed using the resume() method.
- suspend() and resume() method is deprecated in the latest Java version.

# Threads

## Inter-thread Communication Methods

Mechanism by which threads can communicate with each other.

Methods

- wait(): Causes the current thread to wait until another thread invokes notify().
- notify(): Wakes up a single thread that is waiting on this object's monitor.
- notifyAll(): Wakes up all threads that are waiting on this object's monitor.

# Threads

## What is Synchronization?

It is the mechanism that bounds the access of multiple threads to share a common resource hence is suggested to be useful where only one thread at a time is granted the access to run over.

Synchronization

# Threads

## Why Synchronization?

- To prevent thread interference.
- To ensure consistency of shared data.
- To manage access to shared resources.

Synchronization

# Threads

## Synchronization

### Synchronized method

```
public synchronized void method() {
   // synchronized code
}


public synchronized static void staticMethod() {
   // synchronized code
}
```

# Threads

## Synchronized Block

Synchronization

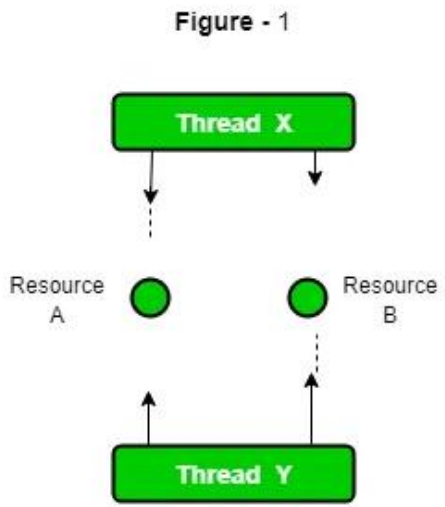
```
public void method() {
    synchronized(this) {
        // synchronized code
    }
}
```

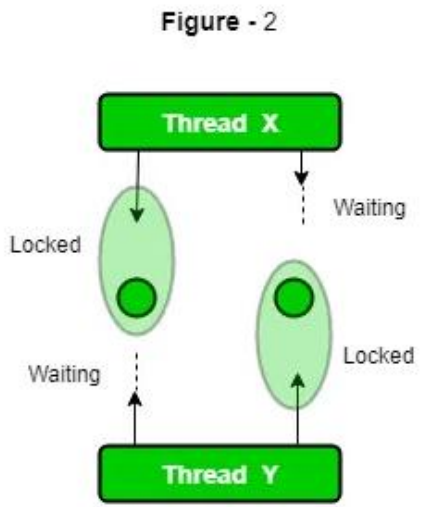
# Threads

## Lock

More flexible locking mechanism than synchronized.

### Reentrant Lock

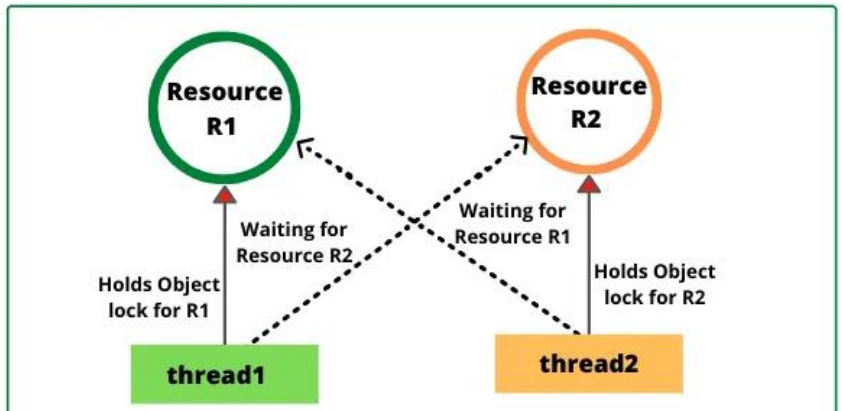


# Threads

## Lock

Dead Lock




Here, both threads are waiting for each other to unlock resources R1 and R2, but thread2 cannot release lock for resource R2 until it gets hold of resource R1.

Fig: Example of Deadlock condition

# Threads

## Lock

### Dead Lock

- Is two threads, each waiting for a lock from the other
- Is not detected or avoided
- Can be avoided by:

  Deciding on the order to obtain locks

  Adhering to this order throughout

  Releasing locks in reverse order

# Threads

## Daemon

Daemon Thread

Threads keep on running even after main thread ends

- Means that the VM still ‘on the air’ until the last thread dies
- To kill a thread when system exits it must be a daemon
- Thread can be set to behave as daemon via *setDaemon(boolean)*
- Thread can be checked via *isDaemon()*

Garbage collection is a daemon thread

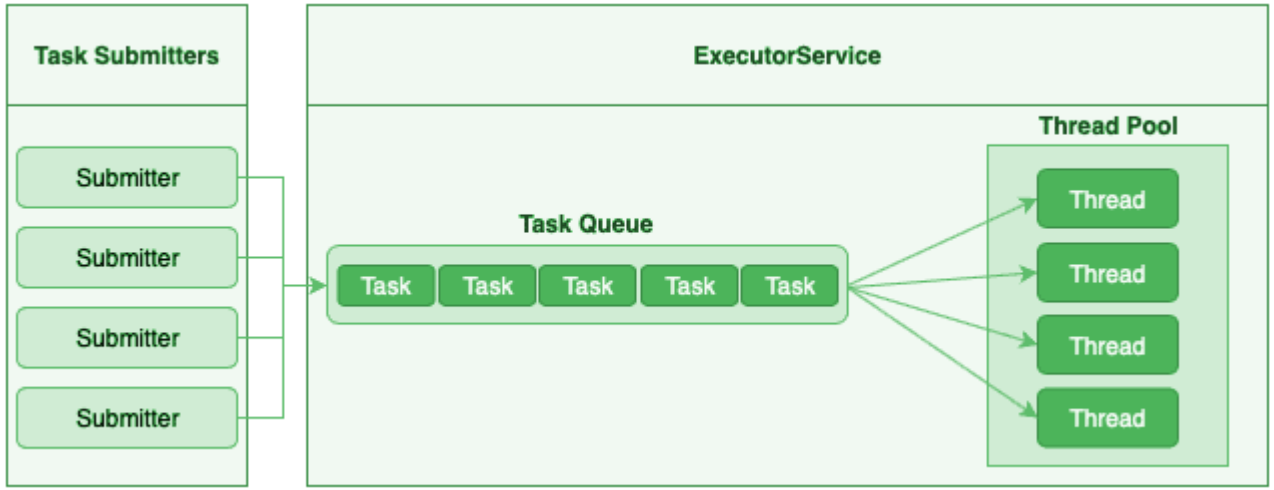
therefore, doesn’t last after system exit

That’s why sometimes object may never get the finalize() call

# Threads

## Executor

Executor



# Threads

Thank You !!