

Object Oriented Programming in Java

Er.Sital Prasad Mandal

BCA- 2nd sem

Mechi Campus

Bhadrapur, Jhapa, Nepal

(Email : info.sitalmandal@gmail.com)

<https://ctaljava.blogspot.com/>



Text Book

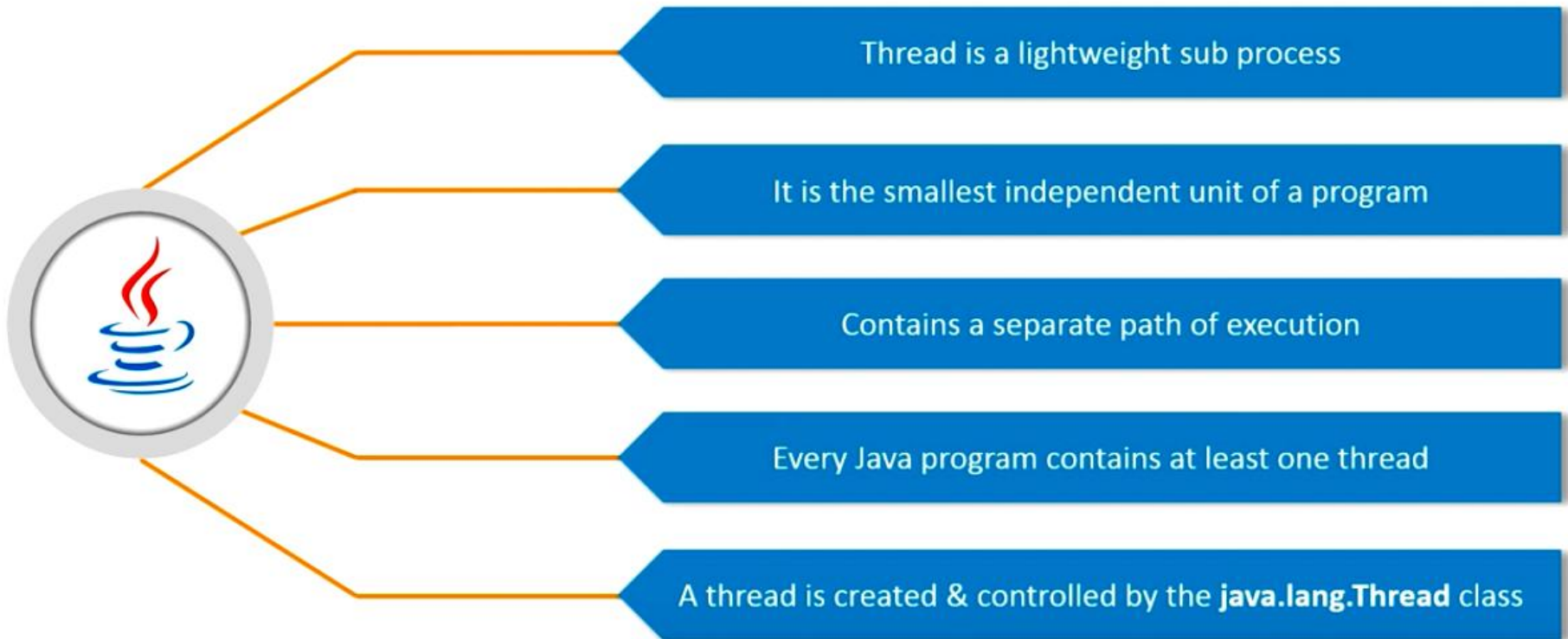
1. Deitel & Dietel. -Java: How to-program-. 9th Edition. TearsorrEducation. 2011, ISBN: 9780273759168
2. Herbert Schildt. "Java: The CoriviaeReferi4.ic e 61 Seventh Edition. McGraw -Hill 2006, 1SBN; 0072263857

7. Threads



7. Threads

Threads



Threads

- 1. Create/Instantiate/Start New Threads:**
 - i. Extending java.lang.Thread**
 - ii. Implementing java.lang.Runnable Interface**
- 2. Understand Thread Execution**
- 3. Thread Priorities**
- 4. Synchronization**
- 5. Inter-Thread Communication**
- 6. Deadlock**

Instantiating a Class

The new operator instantiates a class by allocating memory for a new object.

Note: The phrase "instantiating a class" means the same thing as "**creating an object**";

When you create an object, you are **creating an instance of a class**, therefore "instantiating" a class.

The **new operator** requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

Threads

What are Java Threads?

A thread is a:

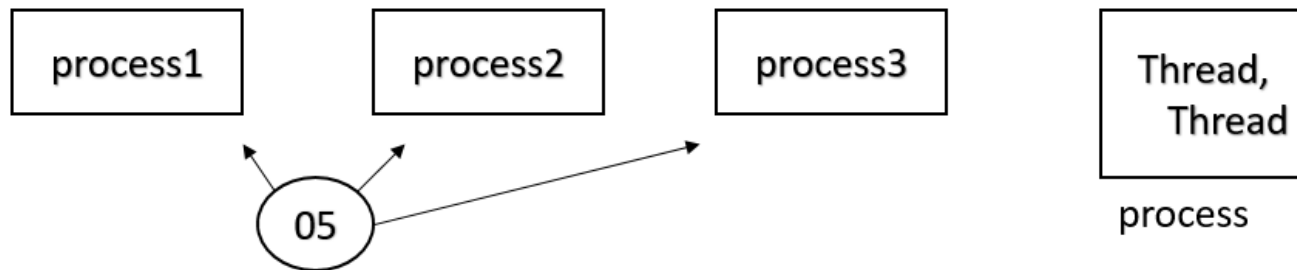
- Facility to allow multiple activities within a single process
- Referred as lightweight process
- A thread is a series of executed statements
- Each thread has its own program counter, stack and local variables
- A thread is a nested sequence of method calls
- Its shares memory, files and per-process state.

*A **thread** in Java is the direction or path that is taken while a program is being executed. Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM or Java Virtual Machine at the starting of the program's execution. At this point, when the main thread is provided, the `main()` method is invoked by the main thread.*

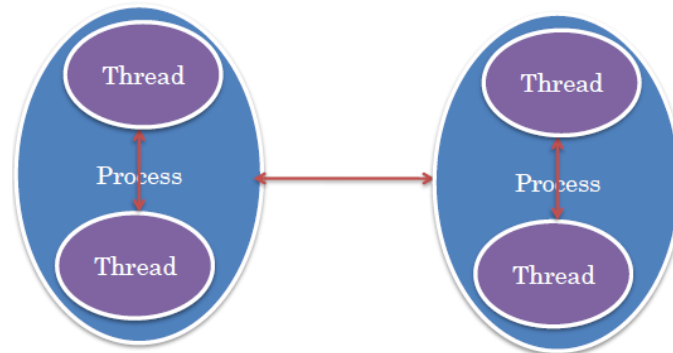
7. Threads

Threads

Multiprocessing and multithreading both are used to achieve multitasking.



- ✓ A **thread** is light-weight where a **process** is heavyweight .
- ✓ for example = A word processor can have one thread running in foreground as an editor and another in the background auto saving the document !

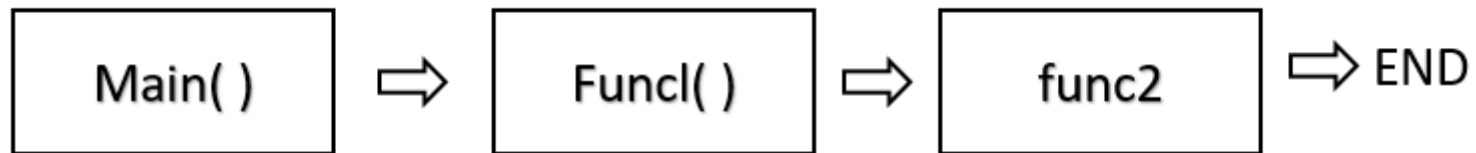


7. Threads

Threads

Flow of control in java

Without threading :



```
class ThreadExample{  
    public static void main(String[] args) {  
        Func1();  
        Func2();  
    }  
}
```

In the above code, you can see that Func1() and Func2() are called inside the main() function. But the execution of Func2() will start only after the completion of the Func1().

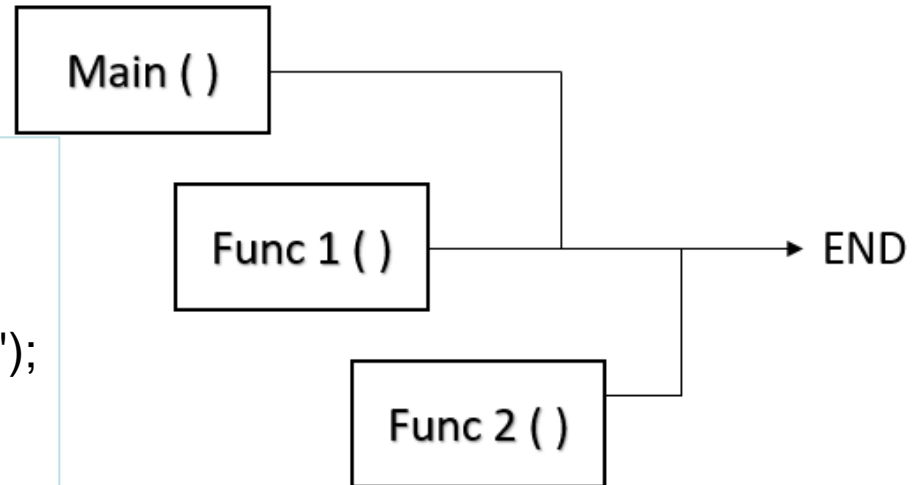
7. Threads

Threads

Flow of control in java

With threading :

```
class Multi extends Thread{  
  
    public void run(){  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]){  
  
        Multi func1=new Multi();  
        func1.start();  
  
        Multi func2=new Multi();  
        func2.start();  
    }  
}
```



Again, Func1() and Func2() are called inside the main function, but none of the two functions is waiting for the execution of the other function. Both the functions are getting executed concurrently.

7. Threads

Multithreading

- ❖ In Java thread programming, multithreading is a process by which we can execute multiple threads simultaneously.
- ❖ This leads to maximum utilization of CPU.
- ❖ Multithreading is used to achieve multitasking.
- ❖ A multithreaded program contains two or more parts that run simultaneously. Each part of such a program is called a thread.
- ❖ It is mostly used to create games and animations.

The path of execution of each thread is different, And they are independent so that even if an exception comes in one thread, it does not affect other threads.

There are two types of thread in this.

First ***user thread*** and second ***daemon thread***.

Daemon threads are used when we want to clean the application and they are used in the background. 'Thread Class' is used to access threads in Java.

7. Threads

Multithreading

Advantage of Multithreading in Java: –

1. It does not block the user because the threads are independent of each other.
2. You can do many tasks at a time by this. Which saves time.
3. All the threads of a process share its resources (such as – memory, data and files etc.). Due to which we do not need to allocate resources to the threads separately.
4. Multithreading reduces idle time of CPU which improves system performance.

Assignment Threads

Whats the need of a thread or why we use Threads?

- To perform asynchronous or background processing.
- Increases the responsiveness of GUI applications.
- Take advantage of multiprocessor systems.
- Simplify program logic when there are multiple independent entities.

7. Threads

Threads

1. Create/Instantiate/Start New Threads

Creating a Threading

There are two ways to create a thread in java

1. By extending thread class (Extending `java.lang.Thread`)
2. By implementing Runnable interface (`java.lang.Runnable` Interface)

Note: The Thread and Runnable are available in the `java.lang.*` package

<https://ctaljava.blogspot.com/>

Threads

1. Extending java.lang.Thread class

In this case, a thread is created by a new class that extends the Thread class, creating an instance of that class. The run() method includes the functionality that is supposed to be implemented by the Thread.

Note:

- The class should extend Java Thread class.
 - The class should override the run() method.
 - The functionality that is expected by the Thread to be executed is written in the run() method.
-
- void start(): Creates a new thread and makes it runnable.
 - void run(): The new thread begins its life inside this method.

Threads

1. Extending java.lang.Thread class

In this case,

- The class should extend Java Thread class, creating an instance of that class.
 - The class should override the run() method.
 - The functionality that is expected by the Thread to be executed is written in the run() method.
- void start(): Creates a new thread and makes it runnable.
 - void run(): The new thread begins its life inside this method.

```
class MyThread extends Thread{  
    @Override  
    public void run(){  
        //code that we want to get executed on running the thread  
    }  
}
```


Threads

1. Extending java.lang.Thread class

In this case,

- The class should extend Java Thread class, creating an instance of that class.
 - The class should override the run() method.
 - The functionality that is expected by the Thread to be executed is written in the run() method.
-
- void start(): Creates a new thread and makes it runnable.
 - void run(): The new thread begins its life inside this method.

```
class MyThread extends Thread{
    @Override
    public void run(){
        //code that we want to get executed on running the thread
    }
}
```

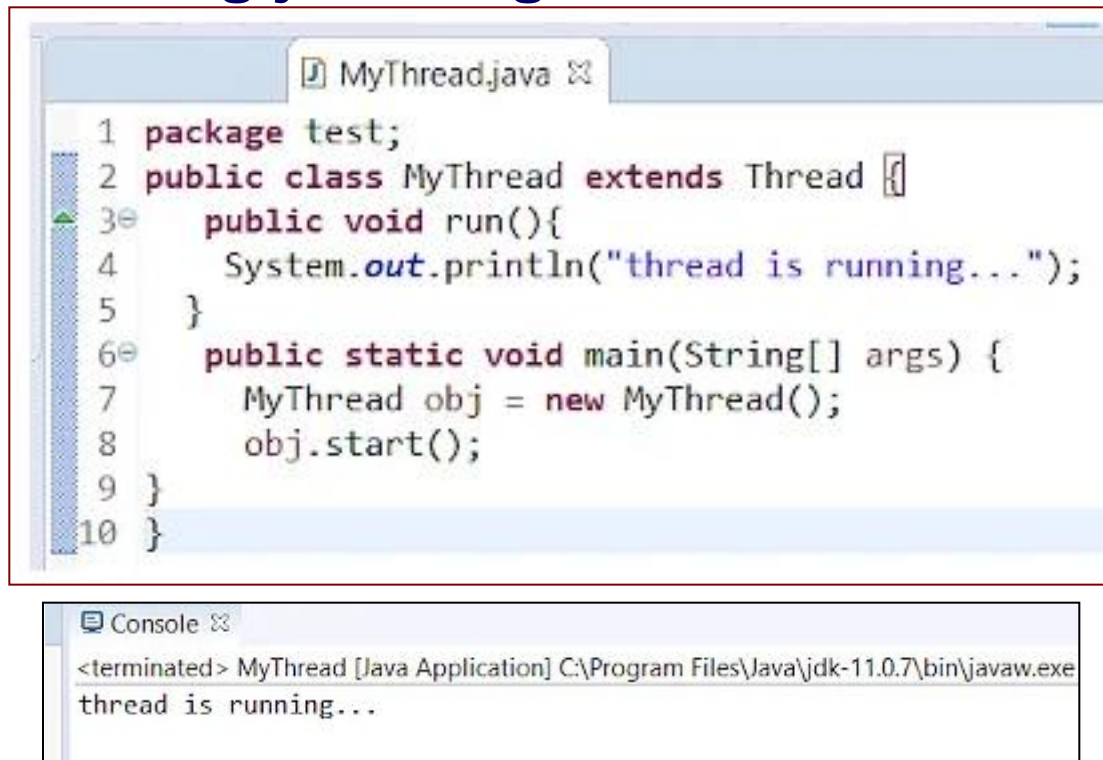
- *In the above code, we're first inheriting the Thread class and then overriding the run() method.*
- *The code you want to execute on the thread's execution goes inside the run() method.*

7. Threads

<https://ctaljava.blogspot.com/>

Threads

1. Extending java.lang.Thread class



```
1 package test;
2 public class MyThread extends Thread {
3     public void run(){
4         System.out.println("thread is running...");
5     }
6     public static void main(String[] args) {
7         MyThread obj = new MyThread();
8         obj.start();
9     }
10 }
```

Console

```
<terminated> MyThread [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe
thread is running...
```

- In order to execute the thread, the start() method is used. start() is called on the object of the MyThread class.
- It automatically calls the run() method, and a new stack is provided to the thread. So, that's how you easily create threads by extending the thread class in Java.

7. Threads

Threads

2. By Implementing Runnable interface (java.lang.Runnable Interface)

This is the easy method to create a thread among the two.

Steps To Create A Java Thread Using Runnable Interface:

1. Create a class and implement the Runnable interface by using the **implements** keyword.
2. Override the run() method inside the implementer class.
3. Create an object of the implementer class in the main() method.
4. Instantiate the Thread class and **pass the object** to the Thread **constructor**.
5. Call start() on the thread. start() will call the run() method.

7. Threads

Threads

2. By Implementing Runnable interface (java.lang.Runnable Interface)

```
package test;  
public class MyThread implements Runnable {  
    public void run(){  
        System.out.println("thread is running..");  
    }  
    public static void main(String[] args) {  
        Thread t = new Thread(new MyThread());  
        t.start();  
    }  
}
```

Console

```
<terminated> MyThread [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe  
thread is running...
```

Threads

2. By Implementing Runnable interface (java.lang.Runnable Interface)

```
classs t1 implements Runnable{
    @Override
    public void run(){
        System.out.println("Thread is running");
    }
}

public class ClassName{
    public static void main(String[] args) {
        t1 obj1 = new t1();
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

1. Class t1 is implementing the Runnable interface.
2. Overriding of the run() method is done inside the t1 class.
3. In the main() method, obj1, an object of the t1 class, is created.
4. The constructor of the Thread class accepts the Runnable instance as an argument, so obj1 is passed to the constructor of the Thread class.
5. Finally, the start() method is called on the thread that will call the run() method internally, and the thread's execution will begin.

7. Threads

Runnable Interface Vs Extending Thread Class :

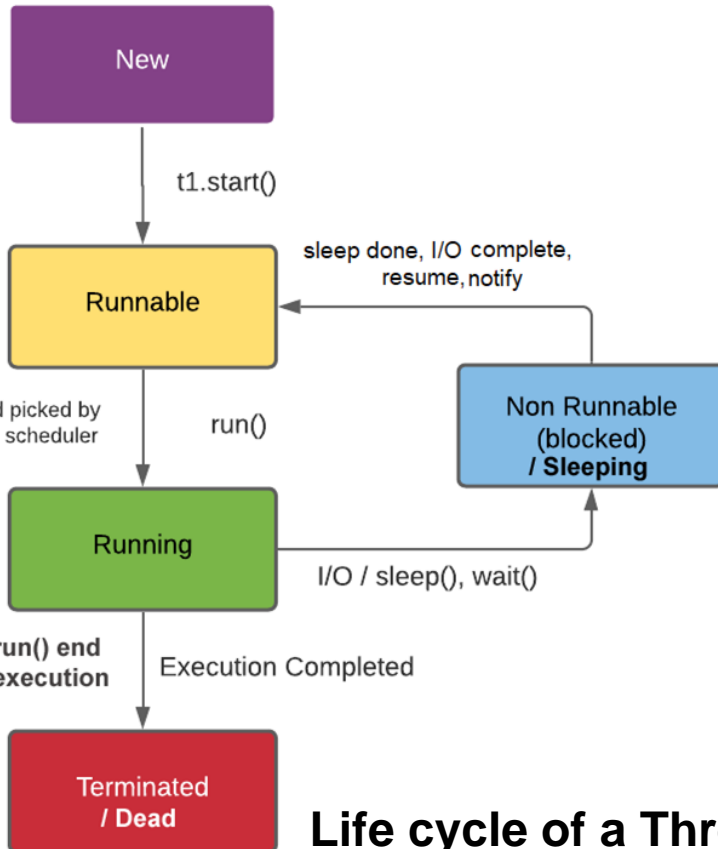
1. As multiple inheritance is not supported in Java, it is impossible to extend the Thread class if your class had already extended some other class.
2. While implementing Runnable, we do not modify or change the thread's behavior.
3. More memory is required while extending the Thread class because each thread creates a unique object.
4. Less memory is required while implementing Runnable because multiple threads share the same object.

7. Threads

Threads

Understand Thread Execution

Thread t1= new Thread()



The Life Cycle of a Thread in Java refers to the state transformations of a thread that begins with its birth and ends with its death.

The `start()` creates the system resources, necessary to run the thread, schedules the thread to run, and calls the thread's `run()`.

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

- If `sleep()` is invoked.
- The thread calls the `wait()`.
- The thread is blocking on I/O.

A thread dies naturally when the `run()` exits.

Life cycle of a Thread (Thread States)

{life cycle events of a thread}

<https://ctaljava.blogspot.com/>

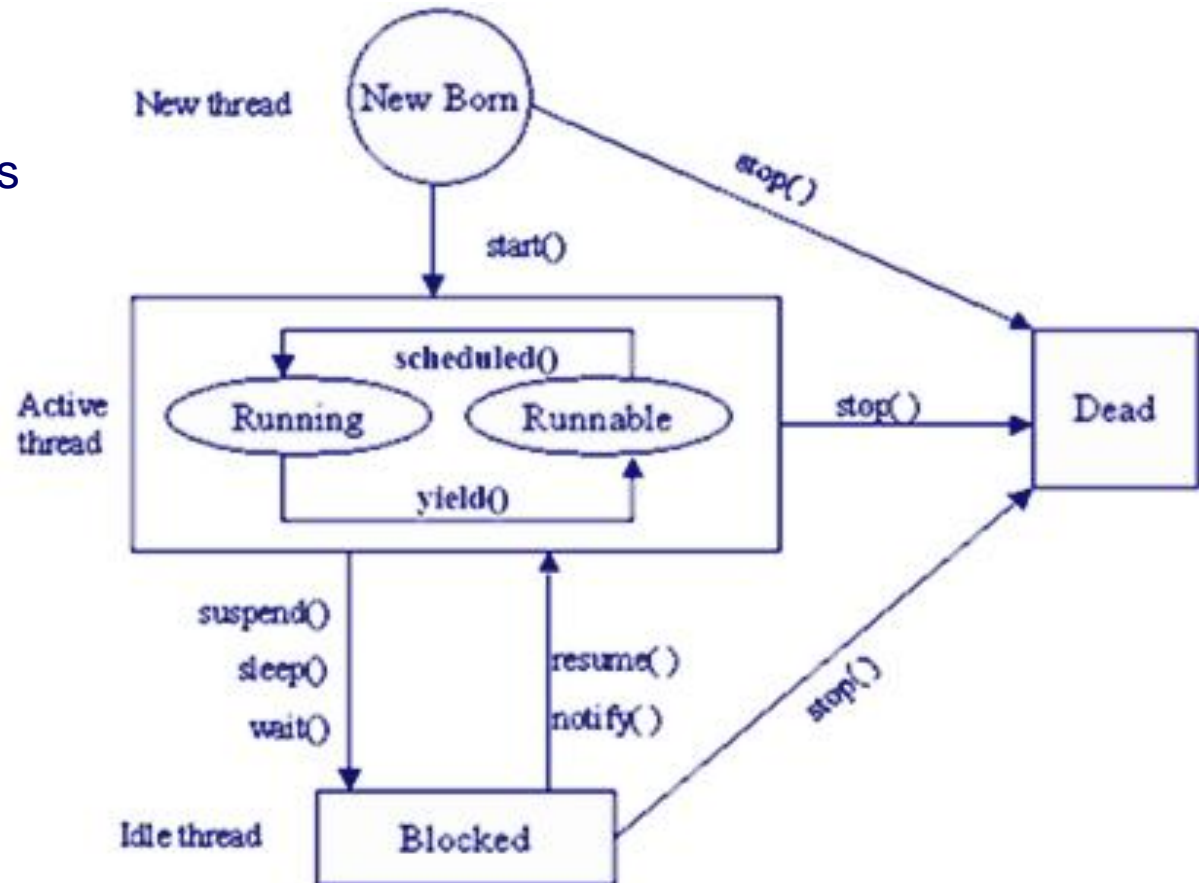
7. Threads

Threads

Understand Thread Execution

There are basically 5 stages in the lifecycle of a thread, as given below:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable)
5. Terminated(Dead)



Phases of thread life cycle

7. Threads

Threads

1. New - The thread is in the new state if you create an instance of Thread class but before the invocation of start() method. In this state, the thread is also known as the born thread.
2. Runnable - After invocation of start() & before it is selected to be run by the scheduler.
3. Running - The thread is in running state *if the thread scheduler has selected it.*
4. Non-runnable - This is the state when the *thread is still alive* but is currently *not eligible to run.*
5. Terminated - thread is in a terminated or dead state *when its run() method exits.*

7. Threads



Program Life Cycle of Threads

```
public class LifeCycleState extends Thread {  
    public void run () {  
        System.out.println("run method");  
    }  
  
    public static void main ( String[] args ) {  
        LifeCycleState p1 = new LifeCycleState();  
        LifeCycleState p2 = new LifeCycleState();  
        System.out.println("p1 State : "+p1.getState());  
        System.out.println("p2 State : "+p2.getState());  
        p1.start();  
        System.out.println("p1 State : "+p1.getState());  
        System.out.println("p2 State : "+p2.getState());  
        p2.start();  
        System.out.println("p1 State : "+p1.getState());  
        System.out.println("p2 State : "+p2.getState());  
    }  
}
```

p1 State : NEW
p2 State : NEW
p1 State : RUNNABLE
p2 State : NEW
p1 State : RUNNABLE
p2 State : RUNNABLE
run method
run method

7. Threads

<https://ctaljava.blogspot.com/>

Assignment Threads

New

A new thread begins its life cycle in this state & remains here until the program starts the thread. It is also known as a **born thread**.

Runnable

Once a newly born thread starts, the thread comes under runnable state. A thread stays in this state is until it is executing its task.

Running

In this state a thread starts executing by entering `run()` method and the `yield()` method can send them to go back to the Runnable state.

Waiting

A thread enters this state when it is temporarily in an inactive state i.e it is still alive but is not eligible to run. It can be in waiting, sleeping or blocked state.

Terminated

A runnable thread enters the terminated state when it completes its task or otherwise terminates.

7. Threads

Assignment Threads

Thread Scheduling

- ❖ Execution of multiple threads on a single CPU, in some order, is called scheduling.
- ❖ In general, the runnable thread with the highest priority is active (running).
- ❖ Java is priority-preemptive algorithm (*process currently under execution is stopped*)
 - If a high-priority thread wakes up, and a low-priority thread is running
 - Then the high-priority thread gets to run immediately
- ❖ Allows on-demand processing
- ❖ Efficient use of CPU

7. Threads

Assignment Threads

Thread Scheduling Types of scheduling

1. Waiting and Notifying

- Waiting [wait()] and notifying [notify(), notifyAll()] provides means of communication between threads that synchronize on the same object.
- wait(): when wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately and moves the thread to the wait state.
- notify(): This wakes up threads that called wait() on the same object and moves the thread to ready state.
- notifyAll(): This wakes up all the threads that called wait() on the same object.

2. Running and Yielding

- Yield() is used to give the other threads of the same priority a chance to execute i.e. causes current running thread to move to runnable state.

3. Sleeping and Waking up

- Sleep() is used to pause a thread for a specified period of time i.e. moves the current running thread to Sleep state for a specified amount of time, before moving it to runnable state. Thread.sleep(no. of milliseconds);

Thread Priorities

- ❖ The number of services assigned to a given thread is referred to as its priority.
- ❖ Any thread generated in the JVM is given a priority.
- ❖ The priority scale runs from 1 to 10.
- ❖ To set these priority, there are two methods `getPriority()` and `setPriority()` in Java.
 - ✓ 1 is known as the lowest priority.
 - ✓ 5 is known as (default) standard priority.
 - ✓ 10 represents the highest level of priority.
- The main thread's priority is set to 5 by default, and each child thread will have the same priority as its parent thread.
- We have the ability to adjust the priority of any thread, whether it is the ***main thread or a user-defined thread***.

It is advised to adjust the priority using the Thread class's constants, which are as follows:

```
Thread.MIN_PRIORITY;  
Thread.NORM_PRIORITY;  
Thread.MAX_PRIORITY;
```

7. Threads

Thread Priorities

Constants for Thread Priorities:-

public static int **MIN_PRIORITY** : This is the minimum priority for the thread. Its value is 1.

public static int **NORM_PRIORITY** : This is the thread's default priority. When no priority is given for the thread, then this priority is set. Its value is 5.

public static int **MAX_PRIORITY** : This is the maximum priority. Its value is 10.

We should use Priority Level $\text{NORM_PRIORITY} - 1 = 4$ and Priority Level $\text{NORM_PRIORITY} + 1 = 6$ respectively for these. To set the priority of a thread, we can use the following statement:

```
myThread.setPriority(Thread.MAX_PRIORITY);
```

Similarly if we want to get the Priority of a Thread, we can set the Priority of that Thread to more or less than other Threads by Incrementing or Decrementing the Priority Level Constant of that Thread. To do this we can use Syntax as follows:

```
myThread.setPriority(myThread.getPriority() + 1 );
```

7. Threads

Thread Priorities

Priority Methods In Java :

1. `setPriority()`:

- This method is used to set the priority of a thread. `IllegalArgumentException` is thrown if the priority given by the user is out of the range [1,10].

Syntax :

```
public final void setPriority(int x)    // x is the priority [1,10]
```

2. `getPriority()`:

- This method is used to display the priority of a given thread.

Syntax :

```
t1.getPriority() // will return the priority of the t1 thread.
```


7. Threads

Thread Priorities

```
public class ThreadPriority extends Thread
{
    public void run ()
    {
        System.out.println ("running thread priority is:" +
            Thread.currentThread ().getPriority ());
    }
    public static void main (String args[])
    {
        ThreadPriority m1 = new ThreadPriority ();
        ThreadPriority m2 = new ThreadPriority ();
        m1.setPriority (Thread.MIN_PRIORITY);
        m2.setPriority (Thread.MAX_PRIORITY);
        m1.start ();
        m2.start ();
    }
}
```

```
<terminated> ThreadPriority [Java Application
running thread priority is:1
running thread priority is:10
```

7. Threads

Java Thread Methods

Join() method In Java :

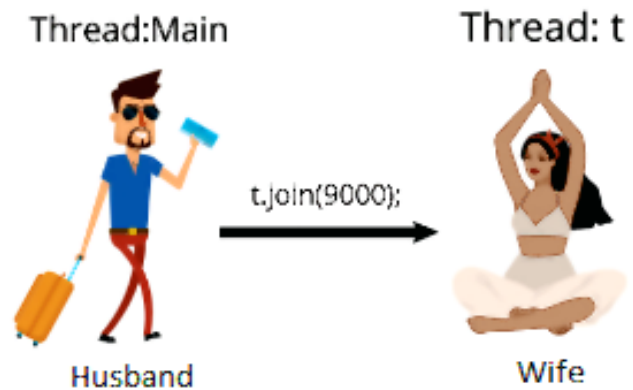
- The join() method in Java allows one thread to wait until the execution of some other specified thread is completed.
- If t is a Thread object whose thread is currently executing, then t.join() causes the current thread to pause execution until t's thread terminates.
- Join() method puts the current thread on wait until the thread on which it is called is dead.

Syntax :

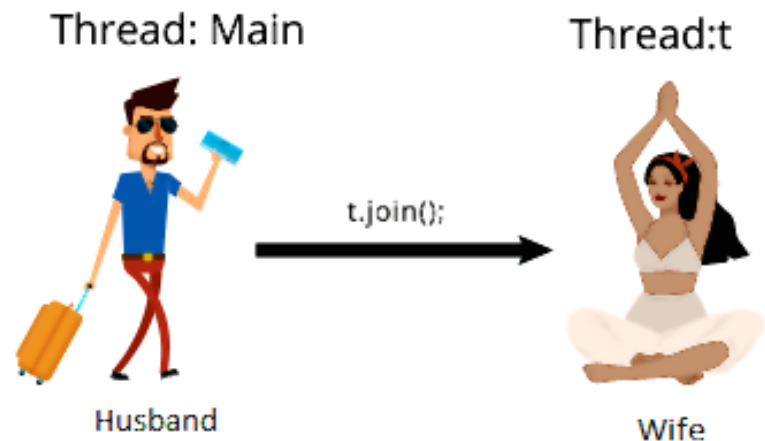
```
public final void join()
```

You can also specify the time for which you need to wait for the execution of a particular thread by using the Join() method. Syntax :

```
public final void join(long millis)
```



Main : Hey 't', get ready in 9 seconds or I'm leaving



Main: Hey 't', I'll wait for you until you get's ready

7. Threads

Java Thread Methods

Interrupt() method :

- A thread in a sleeping or waiting state can be interrupted by another thread with the help of the interrupt() method of the Thread class.
- The interrupt() method throws InterruptedException.
- The interrupt() method will not throw the InterruptedException if the thread is not in the sleeping/blocked state, but the interrupt flag will be changed to true.

Syntax :

Public void interrupt()

Sleep() Method :

- The sleep() method in Java is useful to put a thread to sleep for a specified amount of time.
- When we put a thread to sleep, the thread scheduler picks and executes another thread in the queue.
- Sleep() method returns void.
- sleep() method can be used for any thread, including the main() thread also.

Syntax :

public static void sleep(long milliseconds)throws InterruptedException

public static void sleep(long milliseconds, int nanos)throws InterruptedException

7. Threads

Java Thread Methods

```
import java.io.*;
import java.lang.Thread;
public class cwh {
    public static void main(String[] args)
    {
        try {
            for (int i = 1; i <=5; i++) {
                Thread.sleep(2000);
                System.out.println(i);
            }
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Output :

1
2
3
4
5

In the above example, the main() method will be put to sleep for 2 seconds every time the for loop executes.

<https://ctaljava.blogspot.com/>

Threads

1. Synchronization

Why use Synchronization in Java?

If you start with at least two threads inside a program, there might be a chance when multiple threads attempt to get to the same resource. It can even create an unexpected outcome because of concurrency issues.

Syntax:

```
synchronized(objectidentifier)
{
    // Access shared variables and other shared resources;
}
```

As Java is a multi_threaded, thread synchronization has a lot of importance in Java as multiple threads execute in parallel in an application.

Threads

1. Synchronization

Why use Synchronization in Java?

If you start with at least two threads inside a program, there might be a chance when multiple threads attempt to get to the same resource. It can even create an unexpected outcome because of concurrency issues.

We use keywords “**synchronized**” and “**volatile**” to achieve Synchronization in Java

Syntax:

```
synchronized(objectidentifier)
{
    // Access shared variables and other shared resources;
}
```

As Java is a multi_threaded, thread synchronization has a lot of importance in Java as multiple threads execute in parallel in an application.

7. Threads

Synchronization

In a multithreading environment where multiple threads are involved, there are bound to be clashes when more than one thread tries to get the same resource at the same time. These clashes result in “*race condition*” and thus the program produces unexpected results.

There is a need to synchronize the action of multiple threads. This can be implemented using a concept called **Monitors**.

- Each *object in Java* is associated with a monitor, which a thread can lock or unlock.
- Only one thread at a time may hold a lock on a monitor.
- *Java* programming language provides a very handy way of creating threads and synchronizing their task by using the **Synchronized** blocks.
- It also keeps the shared resources within this particular block.

Most of the time, concurrent access to shared resources in Java may introduce errors like “Memory inconsistency” and “thread interference”. To avoid these errors we need to go for synchronization.

Synchronization

Introduction

Synchronization in java is the capability to control the access of multiple threads to any shared resource. In the Multithreading concept, multiple threads try to access the shared resources at a time to produce inconsistent results. The synchronization is necessary for reliable communication between threads.

Why we use Synchronization

- Synchronization helps in preventing thread interference.
- Synchronization helps to prevent concurrency problems.

Types of Synchronization

Synchronization is classified into two types

1. Process Synchronization
2. Thread Synchronization

Synchronization

Introduction

Synchronization in java is the capability to control the access of multiple threads to any shared resource. In the Multithreading concept, multiple threads try to access the shared resources at a time to produce inconsistent results. The synchronization is necessary for reliable communication between threads.

Why we use Synchronization

- Synchronization helps in preventing thread interference.
- Synchronization helps to prevent concurrency problems.

Types of Synchronization

Synchronization is classified into two types

1. **Process Synchronization**
2. Thread Synchronization (**Inter-Thread Communication**)

7. Threads

Synchronization

Process based multitasking:

Executing several tasks simultaneously where each task **is a separate independent process** such type of multitasking is called process based multitasking.

Example:

- ❑ While typing a java program in the editor we can able to listen mp3 audio songs at the same time we can download a file from the net all these tasks are independent of each other and executing simultaneously and hence it is Process based multitasking.
- ❑ This type of multitasking is best suitable at "os level".

Synchronization

Thread Based Multitasking:

Executing several tasks simultaneously where each task is a separate independent **part of the same program**, is called Thread based multitasking.

And each independent part is called a "Thread".

- This type of multitasking is best suitable for "programatic level".
- Java provides in built support for multithreading through a rich API (Thread, etc)

The main important application areas of multithreading are:

- To implement multimedia graphics.
- To develop animations, video games etc.
- To develop web and application servers

Whether it is Process based or Thread based the main objective of multitasking is to improve performance of the system by reducing response time.

7. Threads

Synchronization

<https://ctaljava.blogspot.com/>

```
class Display {  
    public void wish ( String name ) {  
        for (int i = 0; i < 5; i++) {  
            System.out.println("good morning:" + name);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
            }  
        }  
    }  
}
```

1

```
class MyThread extends Thread {  
    Display d;  
    String name;  
    MyThread ( Display d, String name ) {  
        this.d = d;  
        this.name = name;  
    }  
    public void run () {  
        d.wish(name);  
    }  
}
```

2

```
class SynchronizedDemo {  
    public static void main ( String[] args ) {  
        Display d1 = new Display();  
        MyThread t1 = new MyThread(d1, "Rajesh");  
        MyThread t2 = new MyThread(d1, "Yuvaraj");  
        t1.start();  
        t2.start();  
    }  
}
```

3

7. Threads

Synchronization

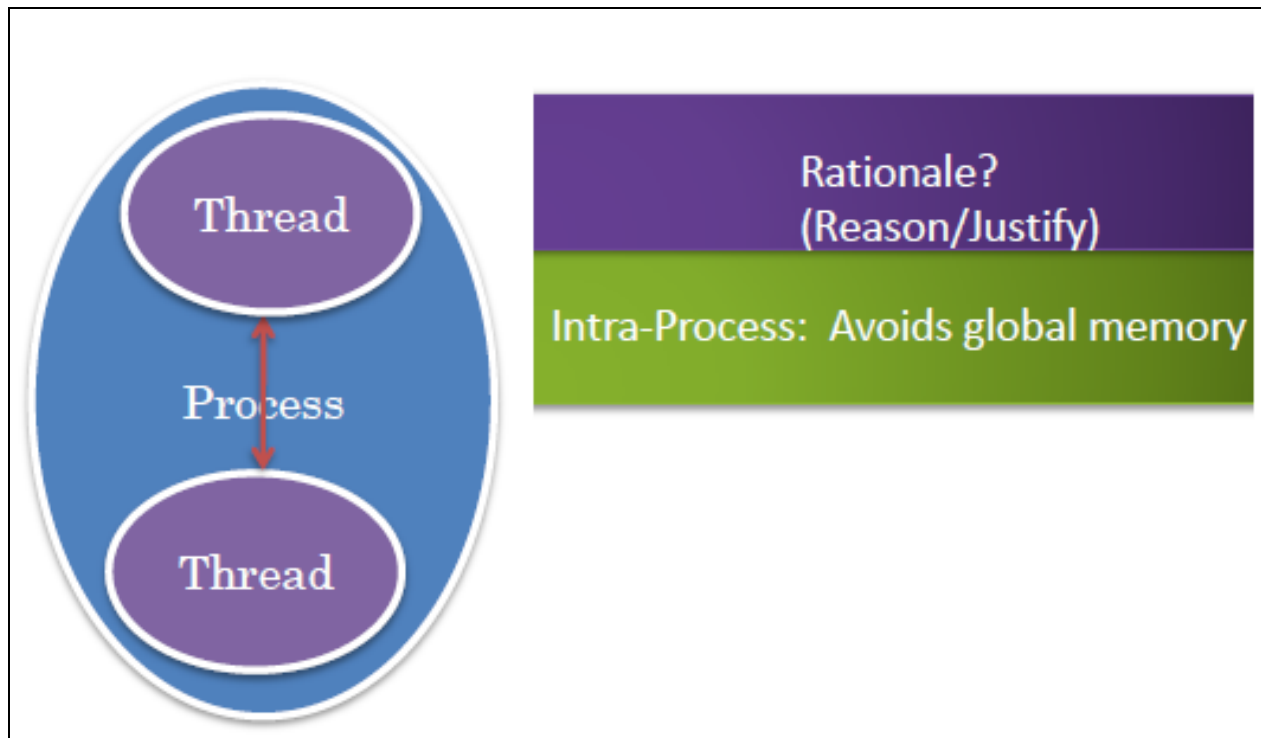
Difference between synchronized keyword and synchronized block

When you use ***synchronized keyword*** with a *method*, it acquires a lock in the object for the entire method. This means that no other thread can use any synchronized method until the current thread that is invoked has finished its execution.

Synchronized block acquires a lock in the object only between parentheses after the synchronized keyword is specified. This means that no other thread can acquire a lock on the already locked object until the block exits. But other threads will be able to access the rest of the code that is present in the method.

7. Threads

Inter-Thread Communication (Cooperation in Java)



7. Threads

Inter-Thread Communication (Cooperation in Java)



- When multiple threads are running inside an application, most of them will need to communicate with each other in some form.
- Threads can communicate each other using `wait()` and `notify()/notifyAll()` methods without any race condition.
- `Wait-and-notify` must be used in conjunction with the synchronized lock to prevent a race condition.
- Methods `wait()`, `notify()` and `notifyAll()` are part of the base class `Object` and not part of `Thread`, as is `sleep()`. Why???

7. Threads

Inter-Thread Communication (Cooperation in Java)

- `sleep()` *does not* release the lock when it is called but method `wait()` *does* release the lock.
- The *only* place you can call `wait()`, `notify()` or `notifyAll()` is within a synchronized method.
- **Restaurant Example:** The waitperson must wait for the chef to prepare a meal. When the chef has a meal ready, the chef notifies the waitperson, who then gets the meal and goes back to waiting.

The chef represents the *producer*, and the waitperson represents the *consumer*.

7. Threads

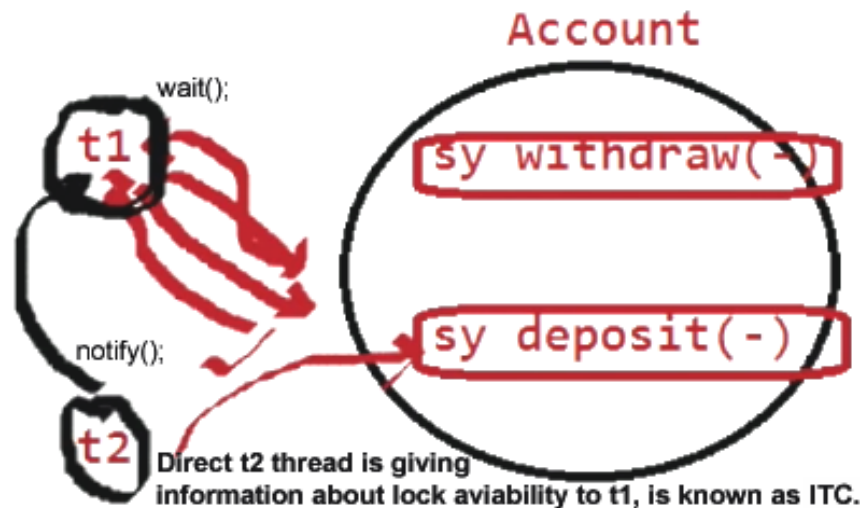
Assignment



- ✓ **Inter-thread communication or Co-operation** is all about allowing synchronized threads to communicate with each other.
- ✓ **Cooperation (Inter-thread communication)** is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object** class:
 1. wait()
 2. notify()
 3. notifyAll()

Inter-Thread Communication (Cooperation in Java)

A thread will provides information to another thread or threads information about lock availability



7. Threads

Inter-Thread Communication Example

```
class Account {  
    public static int balance;  
  
    void displayBalance() {  
        System.out.println("Balance: " + balance);  
    }  
  
    synchronized void withdraw(int amount){  
        if(this.balance < amount){  
            System.out.println("wait to deposit");  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        this.balance = balance - amount;  
        System.out.println( amount + " is withdrawn");  
        displayBalance();  
    }  
  
    synchronized void deposit(int amount){  
        System.out.println("Before Deposit"+ balance);  
        this.balance = balance + amount;  
        System.out.println("After Deposit" + balance);  
        System.out.println( balance + " is deposited");  
        displayBalance();  
        notify();  
    }  
}
```

T1 [withdraw]

T2 [Deposit]

T1 tries to withdraw 1500, but the amount in the bank account is 1000, So T1 will wait() for notify() waits for T2 to deposit more money in the bank account and notify the T1]

Once T1 get the notification from T2, T1 will complete the withdraw.

T2 deposits 900 and notify to T1 by calling notify() method.

7. Threads



Inter-Thread Communication

/ The following Java application shows how the transactions in a bank can be carried out concurrently. */*

```
class Account {
    public static int balance;

    void displayBalance() {
        System.out.println("Balance: " + balance);
    }

    synchronized void withdraw(int amount){
        if(this.balance < amount){
            System.out.println("wait to deposit");
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.balance = balance - amount;
        System.out.println( amount + " is withdrawn");
        displayBalance();
    }

    synchronized void deposit(int amount){
        System.out.println("Before Deposit"+ balance);
        this.balance = balance + amount;
        System.out.println("After Deposit" + balance);
        System.out.println( balance + " is deposited");
        displayBalance();
        notify();
    }
}
```

<https://ctaljava.blogspot.com/>

```
public class IPC_BankTransation {
    public static void main ( String[] args ) {
        Account ABC = new Account();
        ABC.balance = 1000;
        ABC.displayBalance();
        final Account c=new Account();

        new Thread("T1"){
            public void run(){c.withdraw(1500);}
        }.start();

        new Thread("T2"){
            public void run(){c.deposit(900); }
        }.start();
    }
}
```

7. Threads

Inter-Thread Communication (Cooperation in Java)



Quiz 1. Which is NOT a method useful for interThread communication ?

A. wait() B. notify() C. suspend() D. notifyAll()

Quiz 2. During wait(), which method wakes up the thread ?

A. resume() B. sleep() C. notify() D. yield()

7. Threads

Inter-Thread Communication (Cooperation in Java)



Frequently Asked Questions

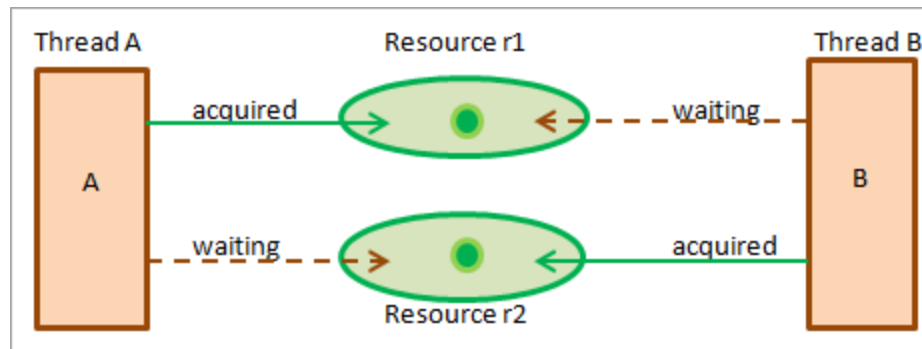
1. Which methods are useful for inter thread communication ?
2. Explain the wait() method
3. Explain the notify() method
4. Define and explain Producer-Consumer problem

7. Threads

Deadlock

Q. What is deadlock in multi threading program?

Deadlock is a situation where two threads are waiting for each other to release lock held by them on resources.



Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a First thread is waiting for an object lock, that is acquired by second thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

7. Threads



Deadlock Create code Fragment

Assumed:

Resource1 = String.class

Resource2 = Object.class

Code called by Thread-1

```
public void run() {  
    synchronized (String.class) {  
        Thread.sleep(100);  
        synchronized (Object.class) {  
        }  
    }  
}
```

Code called by Thread-2

```
public void run() {  
    synchronized (Object.class) {  
        Thread.sleep(100);  
        synchronized (String.class) {  
        }  
    }  
}
```

Thread-1 acquires lock on **String.class [resource1]** and then calls [sleep\(\)](#) method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and **Thread-2** acquires lock on **Object.class[resource2]** then calls sleep() method and **now it waits for Thread-1 to release lock on String.class.**

Conclusion:

Now, Thread-1 is waiting for Thread-2 to release lock on Object.class and Thread-2 is waiting for Thread-1 to release lock on String.class and deadlock is formed.

7. Threads



Deadlock Solved code Fagment

Assumed:

Resource1 = String.class

Resource2 = Object.class

Code called by Thread-1

```
public void run() {  
    synchronized (String.class) {  
        Thread.sleep(100);  
        synchronized (Object.class) {  
        }  
    }  
}
```

Code called by Thread-2

```
public void run() {  
    synchronized (String.class) {  
        Thread.sleep(100);  
        synchronized (Object.class) {  
        }  
    }  
}
```

To solve deadlock just make same parameter as thread 1st Object.class to String.class.

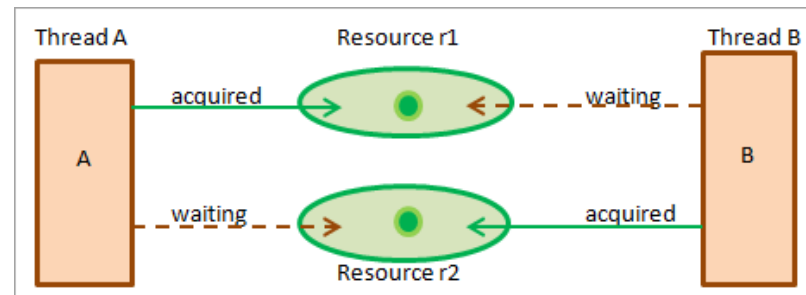
Thread-1 acquires lock on String.class and then calls [sleep\(\)](#) method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and **Thread-2 tries to acquire lock on String.class** but lock is held by Thread-1. Meanwhile, Thread-1 completes successfully. As Thread-1 has completed successfully it releases lock on String.class, Thread-2 can now acquire lock on String.class and complete successfully without any deadlock formation.

Conclusion: No deadlock is formed.

7. Threads

```
public class DeadlockCreation {  
    public static void main ( String[] args ) {  
        // thread_one => Locks shared_res1 then shared_res2  
        Thread t1 = new Thread() {  
            @Override  
            public void run () {  
                synchronized (String.class) {  
                    System.out.println("Thread one: locked shared resource 1");  
                    try {  
                        Thread.sleep(100);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                    synchronized (Object.class) {  
                        System.out.println("Thread one: locked shared resource 2");  
                    }  
                }  
            }  
        };  
  
        Thread t2 = new Thread() {  
            @Override  
            public void run () {  
                synchronized (Object.class) {  
                    System.out.println("Thread two: locked shared resource 2");  
                    try {  
                        Thread.sleep(100);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                    synchronized (String.class) {  
                        System.out.println("Thread two: locked shared resource 1");  
                    }  
                }  
            }  
        };  
  
        t1.start();  
        t2.start();  
    }  
}
```

Thread one: locked shared resource 1
Thread two: locked shared resource 2



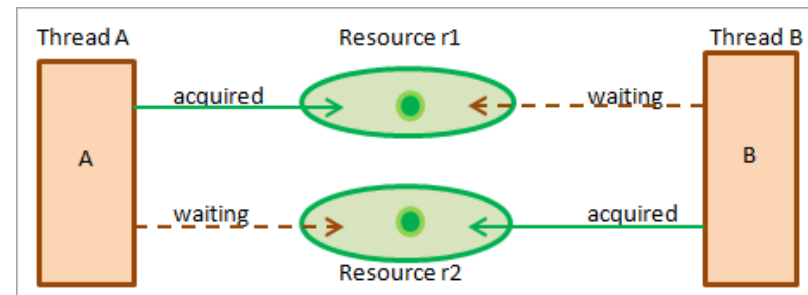
7. Threads

```
public class DeadlockSolution {
    public static void main ( String[] args ) {
        // thread_one => locks shared_res1 then shared_res2
        Thread t1 = new Thread() {
            @Override
            public void run () {
                synchronized (String.class) {
                    System.out.println("Thread one: locked shared resource 1");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (Object.class) {
                        System.out.println("Thread one: locked shared resource 2");
                    }
                }
            }
        };

        Thread t2 = new Thread() {
            @Override
            public void run () {
                synchronized (String.class) {
                    System.out.println("Thread two: locked shared resource 2");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (Object.class) {
                        System.out.println("Thread two: locked shared resource 1");
                    }
                }
            }
        };

        t1.start();
        t2.start();
    }
}
```

Thread one: locked shared resource 1
Thread one: locked shared resource 2
Thread two: locked shared resource 2
Thread two: locked shared resource 1



7. Threads

```
public class DeadLockDemo {
    public static void main(String[] args) {
        //define shared resources
        final String shared_res1 = "Resource1";
        final String shared_res2 = "Resource2";
        // thread_one => Locks shared_res1 then shared_res2
        Thread thread_one = new Thread() {
            public void run() {
                synchronized (shared_res1) {
                    System.out.println("Thread one: locked shared resource 1");

                    try { Thread.sleep(100);} catch (Exception e) {}

                    synchronized (shared_res2) {
                        System.out.println("Thread one: locked shared resource 2");
                    }
                }
            }
        };
        // thread_two=> Locks shared_res2 then shared_res1
        Thread thread_two = new Thread() {
            public void run() {
                synchronized (shared_res2) {
                    System.out.println("Thread two: locked shared resource 2");

                    try { Thread.sleep(100);} catch (Exception e) {}
                    synchronized (shared_res1) {
                        System.out.println("Thread two: locked shared resource 1");
                    }
                }
            }
        };

        //start both the threads
        thread_one.start();
        thread_two.start();
    }
}
```

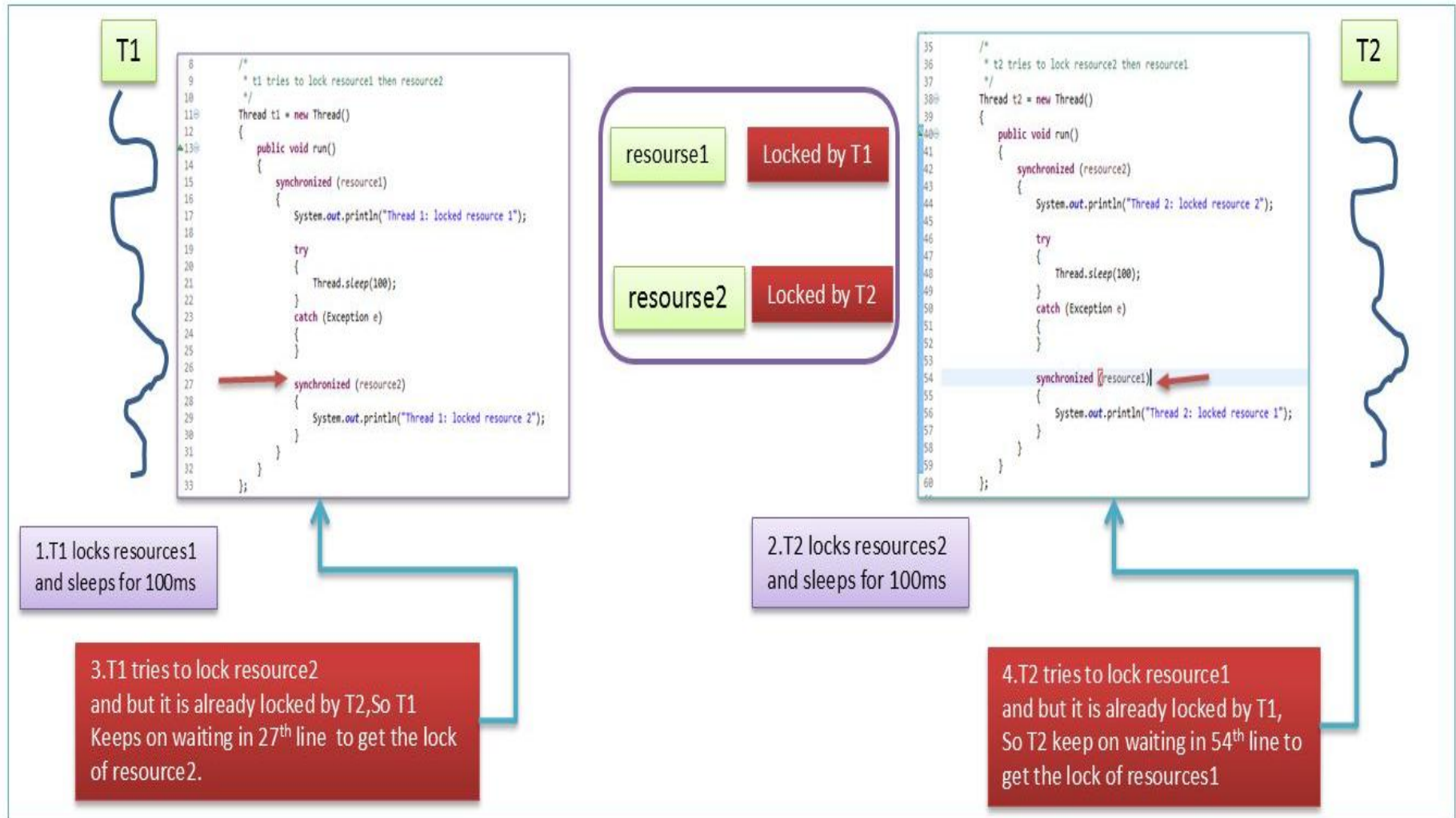
Output:

Thread one: locked shared resource 1

Thread two: locked shared resource 2

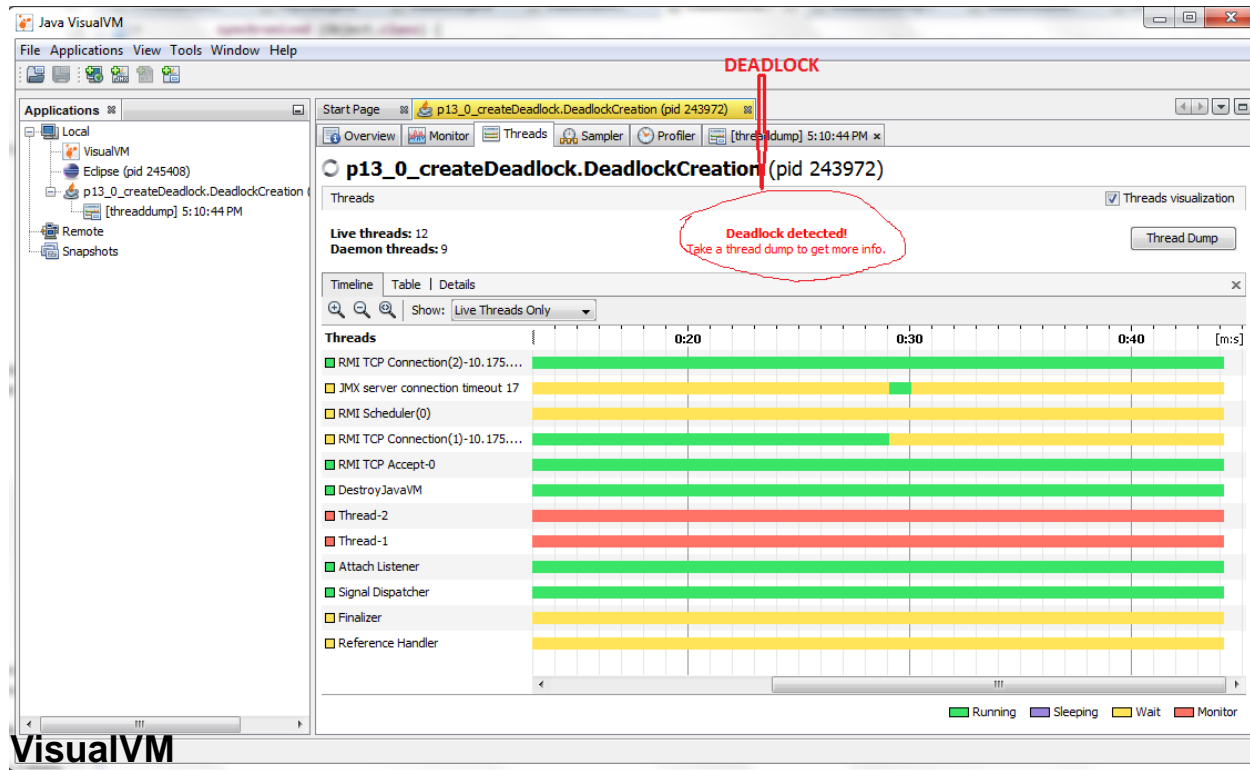
7. Threads

Deadlock



7. Threads

Deadlock



Step: run in VisualVM

Further, we can click **Thread Dump** button to analyze logs >

Before analyzing Thread dump, we must know Few Terminology used, which will help us in analyzing Thread dumps in comprehensive manner >

Thread Dump : Thread gives complete information of threads created in our program and [state transition of Threads](#). i.e. from new to [dead state](#).

Thread Name : Name of the Thread.

<https://ctaljava.blogspot.com/>

7. Threads



Avoid Deadlock

<https://ctaljava.blogspot.com/>

Few important **measures to avoid Deadlock**

1. Lock specific member variables of class rather than locking whole class: We must try to lock specific member variables of class rather than locking whole class.

Example : Let's say we have a class

```
class CustomClass{  
    Integer i;  
    String str;  
}
```

Now rather than locking object of whole CustomClass try to lock specific fields which we want to synchronize.

Avoid such kind of locking :

```
CustomClass customClassObj=new CustomClass();  
synchronized (customClassObj) {  
}
```

Try to use such kind of locking (locking specific member variable of class) :

```
synchronized (customClassObj.str) {  
}
```

Benefit of using such kind of locks is that any other thread can easily operate on unlocked member variable of class and reduce risk of forming deadlock.

7. Threads

<https://ctaljava.blogspot.com/>



Avoid Deadlock

Few important **measures to avoid Deadlock**

2. **Use join() method**: If possible try to use join() method, although it may refrain us from taking full advantage of multithreading environment because threads will start and end sequentially, but it can be handy in avoiding deadlocks.

Now, I am going to provide you with source code to give you a better picture of how join() method can be handy in avoiding deadlock in above used program.

```
public class DeadlockAvoidingUsingJoin {  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread1=new Thread(new A(),"Thread-1");  
        Thread thread2=new Thread(new B(),"Thread-2");  
        thread1.start();  
        thread1.join();  
        thread2.start();  
    }  
}
```

3. **If possible try avoid using nested synchronization blocks.**

<https://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program.html>