

# **JAVA @11**

## Java Thread – Part I

# Objective

After completing this session you will be able to understand,

- What is Multithreading?
- Life cycle of a Thread
- Creating Thread
- Thread Scheduler
- Thread Interruption techniques
- Thread Priority

# Process vs Thread

Before we learnt about threads let's first understand the difference between a thread and a process.

**Process** are executables which run in separate memory space.

**Threads** are small process which run in shared memory space within a process.

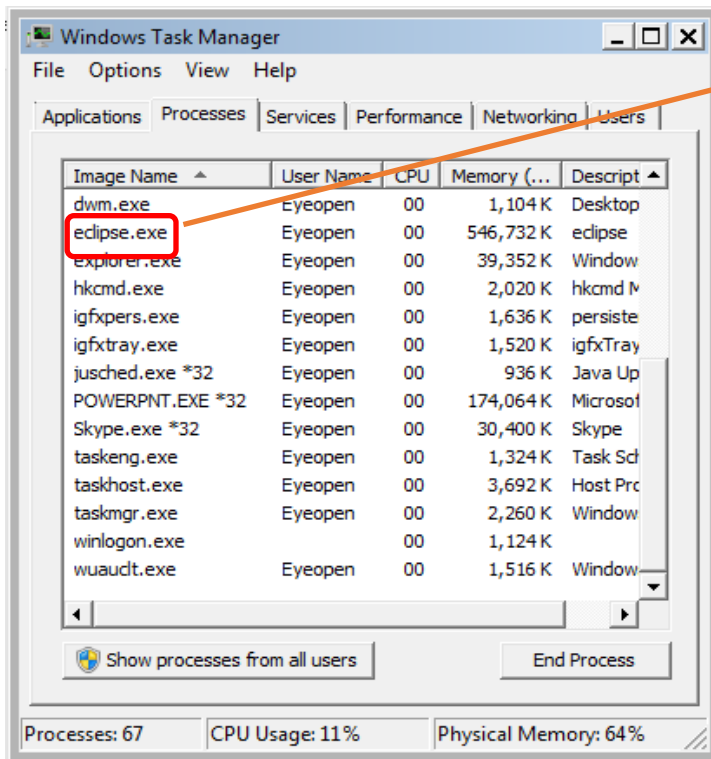
***So in a nutshell Process is a container for Threads, all thread runs inside the process.***



**Still confused let's look at an example to understand it better.**

# Process

Lets consider Eclipse IDE application to understand it better, what happens when you start an eclipse application.



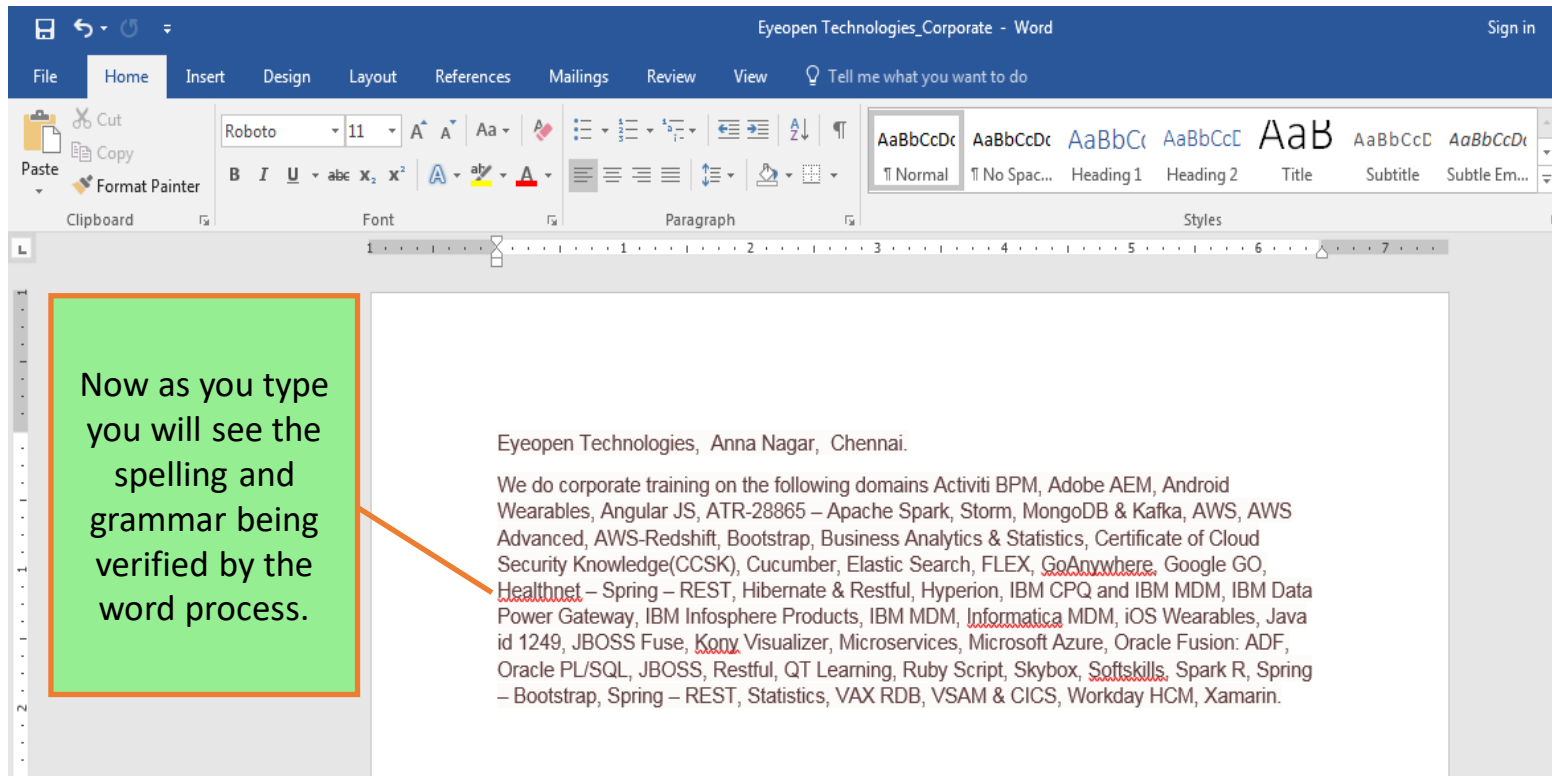
A process for eclipse ide application will be started.

Now lets see what happens when the user starts using word application.

# Thread

The spell check has been implemented as a *thread* within the word.exe process which runs continuously and verifies what you type.

Word.exe is the process and spell check is a thread running inside the process.



# Real time scenario

**Ram** was developing an application where he has a requirement where user can register his profile in the application, assume registration has three steps

**Validate user details** – Takes 3 seconds for execution for each user.

**Validate user Citizenship** - Takes 4 seconds for execution for each user.

Now customer wants to complete the registration process is less than 5 seconds.



Guess how **Mr. Ram** would have achieved it?

He used **Multi Threading**.

Lets see what it is and how it can be implemented in this session.

# Multitasking

## What is Multitasking?

- Refers to a computer's ability to perform multiple jobs concurrently
- More than one program are running concurrently,

Example: In windows you can run Word, power point , media player at the same time. Yu will working on word and in parallel media player might play some music.

# Multithreading

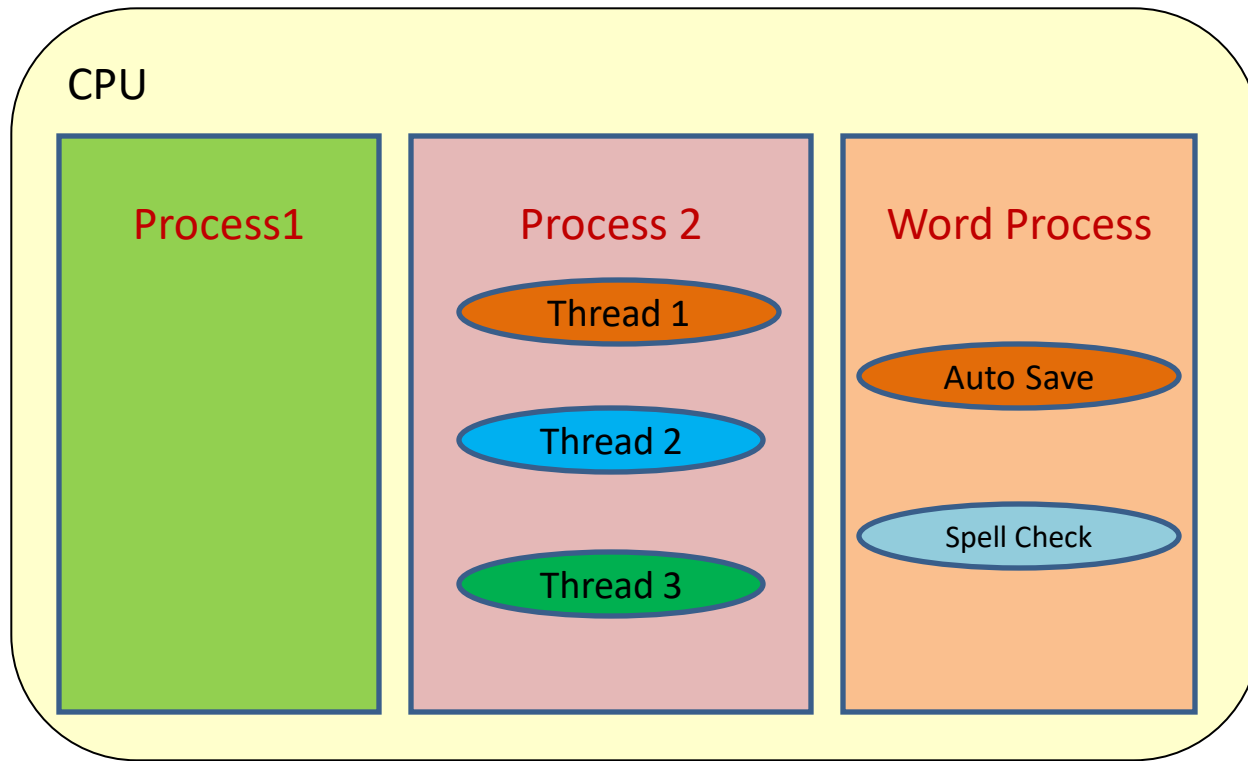
## What is Multithreading?

- A thread is a single sequence of execution within a program/process.
- This refers to multiple threads of control within a single program.
- Each program can run multiple threads of control within it.

Example: Microsoft word process having multiple threads like spell check, auto save etc.



# System Illustration



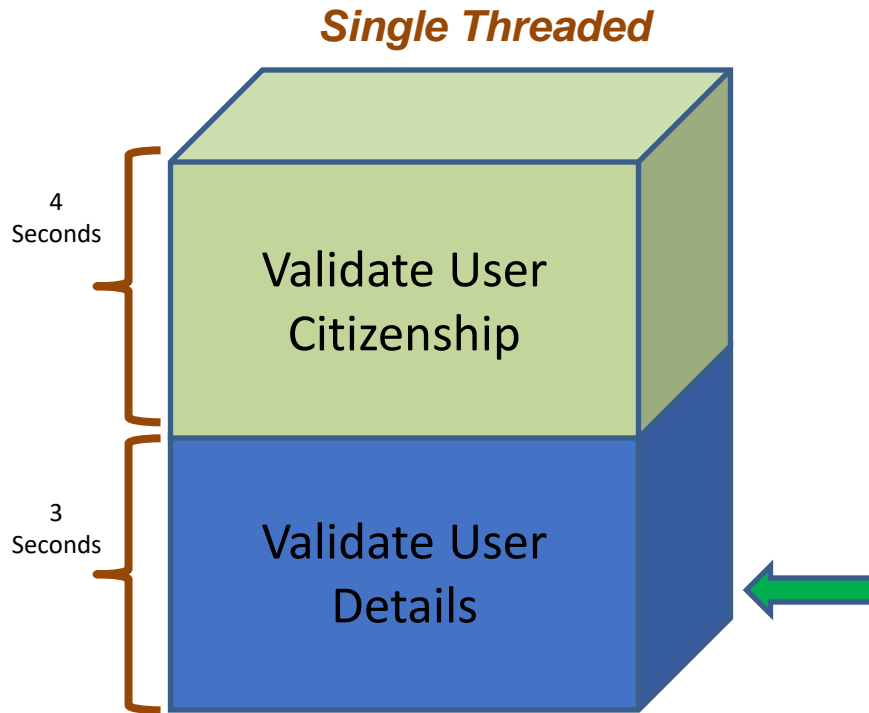
The CPU is running three processes. Process 2 in turn has three threads running inside it.

# Benefits of Multithreading

- To reduce response (execution) time of a process.
- Support Parallel Operation of Functions.
- Increase System Efficiency.
- Requires less overheads compared to Multitasking.

# How Ram Solved the Problem?

Ram implemented two threads for processing the **Validate user details** and **Validate user Citizenship**.



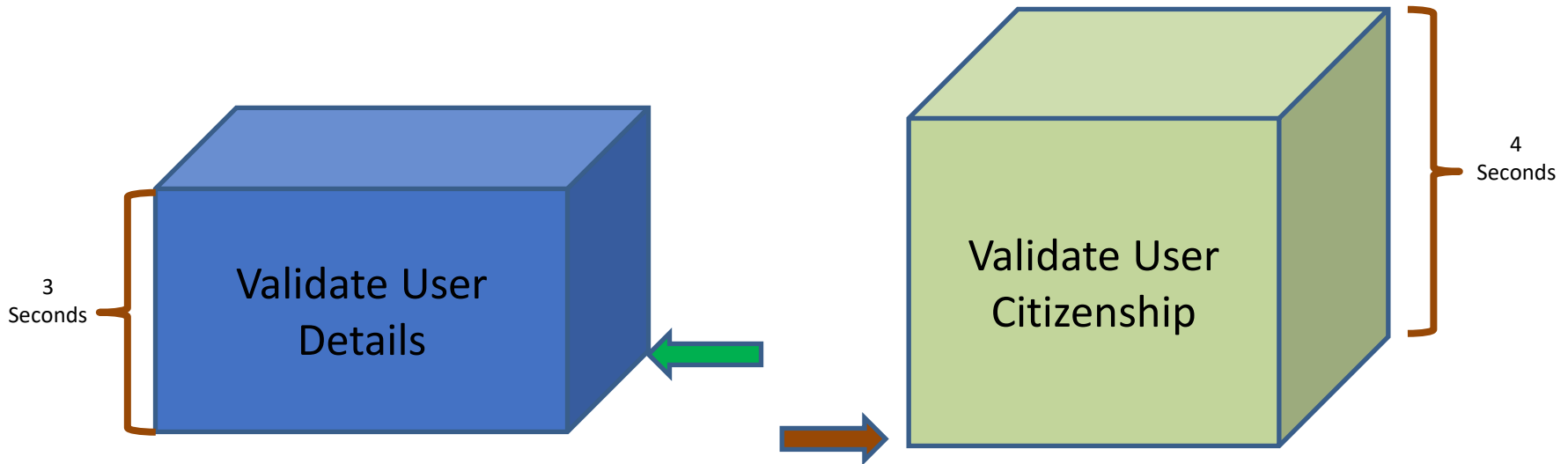
The thread takes 7 seconds for completing the registration process.

Let look how Ram implemented it.

# How Ram Solved the Problem?

Ram implemented two threads one thread for each method **Validate user details** and **Validate user Citizenship**.

*Multi Threaded*



The process will be completed in 4 seconds.  
Since both the threads works in parallel.

# What is an Application Thread?

*When we execute an application,*

- 1. The JVM creates a thread (T1) object which invokes the main() method and starts the application.*
- 2. The thread executes the statements of the program one by one (or) starts other threads.*
- 3. After executing all the statements, the method returns and the thread dies.*

The thread **T1** which is responsible for starting the application by invoking the main method is called “***Application Thread***”.

# Ways of Implementing Threads

**Method 1** : Extend *Thread* Class

**Method 2** : Implement *Runnable* Interface

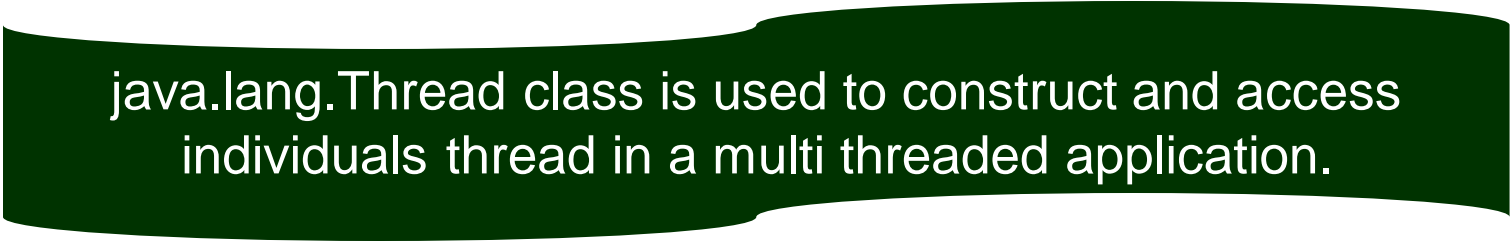
# Using Thread in Java



How To Use  
Threads in  
Java



Using Thread  
class



java.lang.Thread class is used to construct and access individuals thread in a multi threaded application.

# Using Thread in Java

```
public class Thread extends Object implements Runnable {  
    public Thread();  
    public Thread(String name); // Thread name  
    public Thread(Runnable r); // Thread  $\Rightarrow$  r.run()  
    public Thread(Runnable r, String name);  
    public void run();  
    public void start();// begin thread execution  
}
```

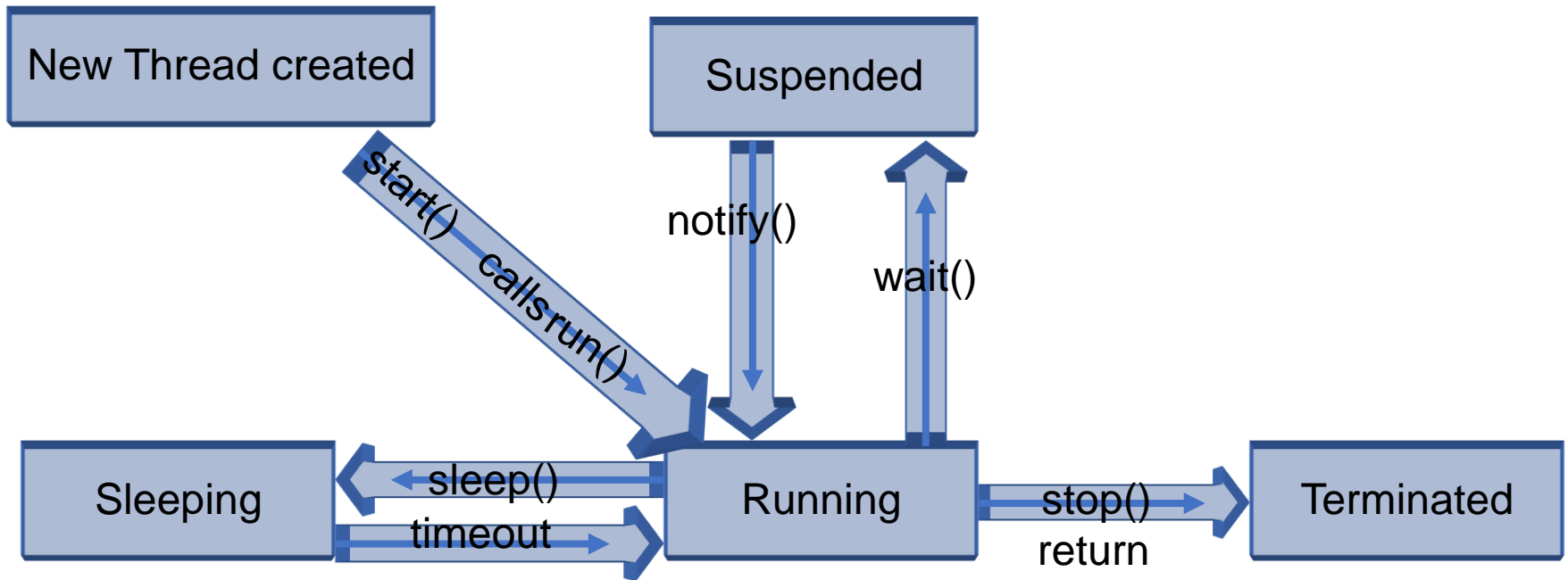


# Thread class methods

Method	Description
<b>void run()</b>	<ul style="list-style-type: none"><li>• The thread logic should be implemented in this method.</li><li>• This method should be overridden in all the Thread class.</li></ul>
<b>void start()</b>	Creates a new thread and invokes run method of the thread.
<b>getName()</b>	Returns the thread's name.
<b>int getPriority()</b>	Returns the thread's priority
<b>boolean isAlive()</b>	Tests if the thread is alive.

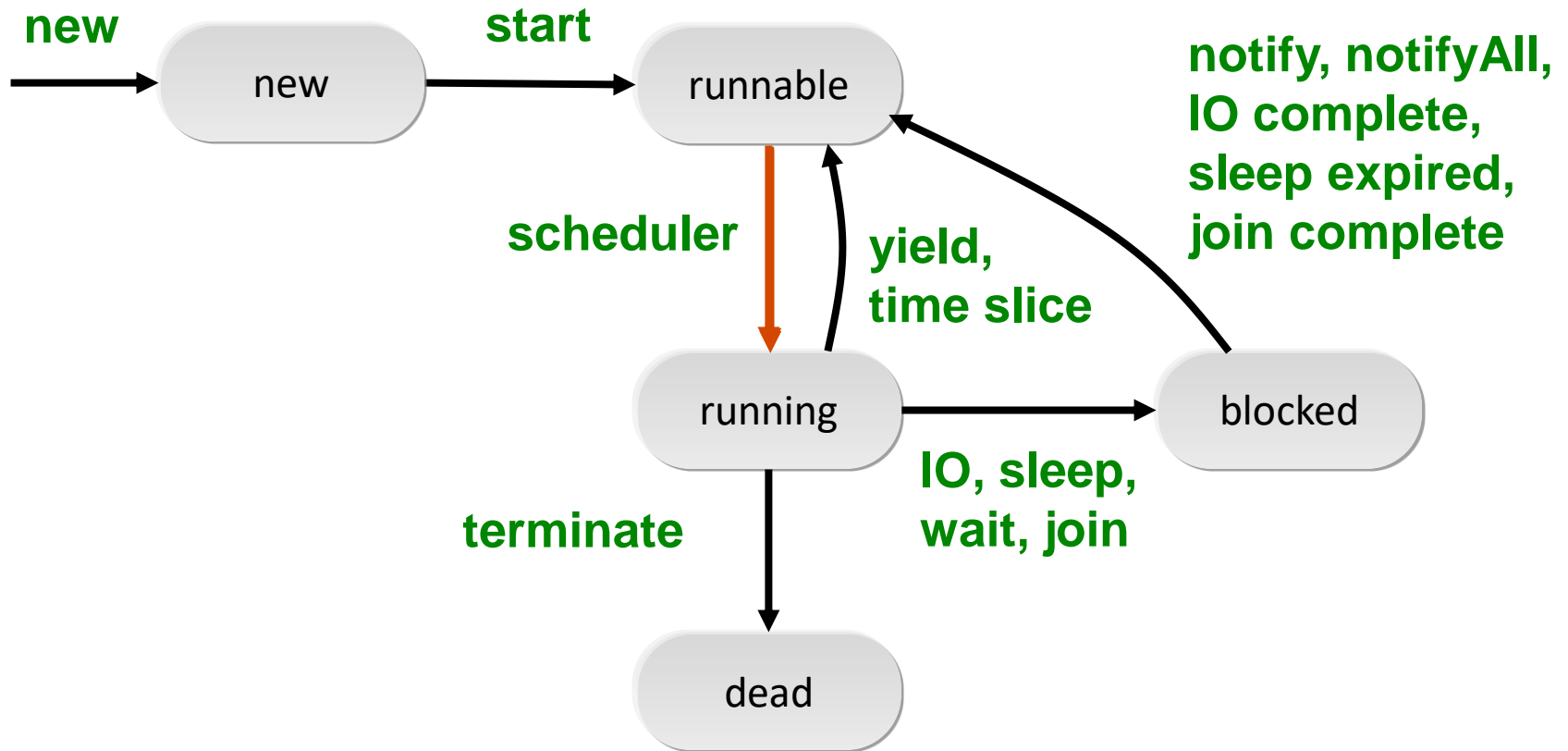
Method	Description
<b>Thread currentThread()</b>	Return a reference to the currently executing thread object
<b>setName(String name)</b>	Sets a name for the thread to be equal to the argument name.
<b>setPriority (int newPriority)</b>	Changes the priority of the thread
<b>static void sleep(long millis)</b>	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
<b>void yield()</b>	Causes the currently executing thread object to temporarily pause and allow other threads of same priority to execute

# Lifecycle of Thread



# Java thread states

## State Diagram



# Example – How to develop a thread?

In this demo we will learn how to

1. Create a Thread by extending the Thread class
2. Override the run method
3. Create a thread object and start the thread using the start method
4. How the following methods operates
  - a. getName()
  - b. setName()
  - c. setPriority()
  - d. getPriority()
  - e. sleep()

## **Components to be developed,**

1. ThreadEX – The Thread class should loop and print values 0...4.
2. ThreadExMain – The main class to execute the Threads

# Solution - Thread

```
public class ThreadEx extends Thread {  
    int i = 0;
```

Class ThreadEx extends Thread class and becomes a Thread class

```
    public ThreadEx(String name) {  
        this.setName(name);
```

Sets the name of the Thread using setName method()

```
    }
```

Overrides  
the run()  
method

Prints the name of the current  
Thread using the getName()  
method

```
    public void run() {  
        for (i = 0; i < 5; i++) {  
            System.out.println("Printing from " + Thread.currentThread().getName()  
                               + " the value of i :: " + i);  
            try {  
                Thread.sleep(300);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

Makes the Thread sleep for 300  
milliseconds. Invoking the sleep may  
cause an InterruptedException to be  
thrown which should be handlers.

# Solution - Thread

```
public class ThreadExMain {  
    public static void main(String args[]) throws InterruptedException {  
        Thread t = Thread.currentThread();  
        System.out.println("The Current Thread is " + t.getName());  
        System.out.println("The Current Thread Priority is " + t.getPriority());  
        t.setName("myThread");  
        t.setPriority(6);  
        System.out.println("The Current Thread is " + t.getName());  
        System.out.println("The Current Thread Priority is " + t.getPriority());  
        ThreadEx ex1 = new ThreadEx("first");  
        ThreadEx ex2 = new ThreadEx("second");  
        System.out.println("The Thread Priority of ex1 is "  
            + ex1.getPriority());  
        System.out.println("The Thread Priority of ex2 is "  
            + ex2.getPriority());  
        ex1.start();  
        ex2.start();  
        for (int i = 0; i < 5; i++) {  
            System.out.println("Printing from : "  
                + Thread.currentThread().getName() + " :: " + i);  
            Thread.sleep(100);  
        }  
    }  
}
```

Gets the priority and name of the main thread

Sets the priority and name of the main thread

Creates two new Thread objects with name "first" and "second"

Starts the Thread by invoking the start () which will invoke the run() method of the Thread class

# Solution - Thread

- Execute the main class – ***ThreadExMain***
- The output will be something as shown below – Output may vary for each execution since Thread execution is based on the underlying operating system.

```
The Current Thread is main
The Current Thread Priority is 5
The Current Thread is myThread
The Current Thread Priority is 6
The Thread Priority of ex1 is 6
The Thread Priority of ex2 is 6
Printing from : myThread :: 0
Printing from first the value of i :: 0
Printing from second the value of i :: 0
Printing from : myThread :: 1
Printing from : myThread :: 2
Printing from first the value of i :: 1
Printing from second the value of i :: 1
Printing from : myThread :: 3
Printing from : myThread :: 4
Printing from second the value of i :: 2
Printing from first the value of i :: 2
Printing from second the value of i :: 3
Printing from first the value of i :: 3
Printing from second the value of i :: 4
Printing from first the value of i :: 4
```

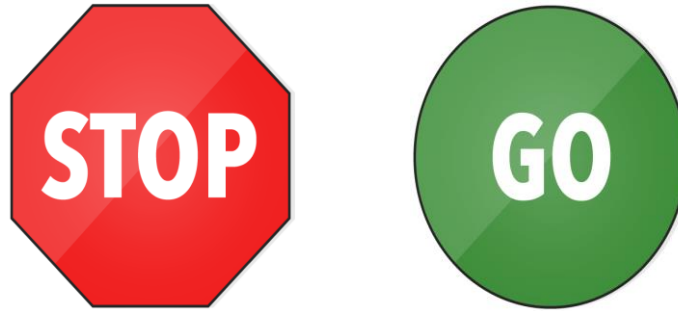


Different output  
displayed during  
different run

```
The Current Thread is main
The Current Thread Priority is 5
The Current Thread is myThread
The Current Thread Priority is 6
The Thread Priority of ex1 is 6
The Thread Priority of ex2 is 6
Printing from : myThread :: 0
Printing from first the value of i :: 0
Printing from second the value of i :: 0
Printing from : myThread :: 1
Printing from : myThread :: 2
Printing from second the value of i :: 1
Printing from first the value of i :: 1
Printing from : myThread :: 3
Printing from : myThread :: 4
Printing from second the value of i :: 2
Printing from first the value of i :: 2
Printing from first the value of i :: 3
Printing from second the value of i :: 3
Printing from first the value of i :: 4
Printing from second the value of i :: 4
```



# Time To Reflect



**Trainees to reflect the following topics before proceeding.**

- What is Multithreading?
- How can we implement threading in Java API?
- Explain about Thread Lifecycle methods in Java API?

Thank you

*You have successfully completed*  
**Thread-1**

**JAVA @11**

Java Thread – Part II

# Objective

After completing this session you will be able to understand,

- How to create thread by implementing runnable interface?
- Comparison between the two method of thread creation
- Learn about thread join method

# Runnable Interface

- Threads can be created by implementing the *Runnable* interface and overriding the run method.
- *Runnable* interface contains only one method – the *run()* method which should be overridden to start a new Thread.

# Implementing Runnable Interface

**Step 1:** Create a class ***ThreadRunnableEx*** by implementing the *Runnable* interface

**Step 2:** Override the ***run()*** method , the logic each thread should execute.

**Step 3:** Create an ***main*** method which can start the thread.

**Step 4:** Inside the main method create an object of the ***ThreadRunnableEx*** class.

**Step 5:** Inside the main method create a *Thread* object using the ***ThreadRunnableEx***. This is done by passing the ***ThreadRunnableEx*** object to the constructor of the Thread class.

**Step 6:** Invoke the start method on the Thread object which in turn calls the *run()* method of the ***ThreadRunnableEx*** class and starts a thread.

# Example – Runnable Interface

**In this demo we will learn how to**

1. Create a Thread by using Runnable interface.
2. Implement the run method.
3. Create a thread object using the Runnable interface,

**Components to be developed,**

1. **ThreadRunnableEx** – Thread implemented using Runnable interface. Each Thread should loop and print values 0...9
2. **RunnableExMain** – The main class to start the Threads

# Solution – Runnable Interface

Create a class named ***ThreadRunnableEx*** implementing the *Runnable* interface

```
public class ThreadRunnableEx implements Runnable {  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("Printing from " +  
                               Thread.currentThread().getName() +  
                               " , the value of i : " + i);  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```

Class ThreadRunnableEx implements *Runnable* interface override the run method to print values between 0..9.



# Solution – Runnable Interface

Create class *RunnableExMain* with a main method which will start the threads.

```
public class RunnableExMain {  
    public static void main(String args[]) throws InterruptedException {  
        ThreadRunnableEx r1 = new ThreadRunnableEx();  
        ThreadRunnableEx r2 = new ThreadRunnableEx();  
        Thread t1 = new Thread(r1, "first");  
        Thread t2 = new Thread(r2, "second");  
        t1.start();  
        t2.start();  
        for (int i = 0; i < 10; i++) {  
            System.out.println(Thread.currentThread().getName() + " :: " + i);  
            Thread.sleep(100);  
        }  
    }  
}
```

This creates two thread to be run

Create the Thread objects using the Runnable objects

Start the two Threads

# Execute the Runnable

- Run the main class – ***ThreadExMain***
- The output will be something as shown below.

Output may vary for each execution since Thread execution depends upon the operating system.

```
main :: 0
Printing from second , the value of i :0
Printing from first , the value of i :0
Printing from first , the value of i :1
main :: 1
Printing from second , the value of i :1
Printing from second , the value of i :2
main :: 2
Printing from first , the value of i :2
Printing from second , the value of i :3
Printing from first , the value of i :3
main :: 3
Printing from first , the value of i :4
Printing from second , the value of i :4
main :: 4
```

Different output  
displayed during  
different run

```
main :: 0
Printing from second , the value of i :0
Printing from first , the value of i :0
main :: 1
Printing from second , the value of i :1
Printing from first , the value of i :1
main :: 2
Printing from second , the value of i :2
Printing from first , the value of i :2
main :: 3
Printing from second , the value of i :3
Printing from first , the value of i :3
main :: 4
Printing from second , the value of i :4
Printing from first , the value of i :4
```

# Runnable vs Thread



Which Method is better? Why?

Implementing Runnable interface is recommended.

## Why Runnable is better?

- If we inherit the *Thread* class for creating threads we cannot extend any other class. *Since Java does not support multiple inheritance*
- By implementing the *Runnable* interface we can make the thread class extend other classes.

# Ram faces another problem

In the previous examples we can see that the main Thread (***Application Thread***) exits before the other child Threads which was originated by it completes execution. This may not be desirable in some situations.

**Example:** Assume tax needs to be calculated for 10 employees using **calculateTax()** method, Tim the programmer decides to speedup the process by spawning 10 thread one thread per employee. Also Tim needs to ensure that the program should exit only after all the ten threads complete the tax calculation.

How can this be solved?  
Tim used Thread **Join** Method.

# What is join method?

join method in Thread class,

- ***join()*** is used to ensure that the application main thread will complete only after all the thread spawned by it inside the process completes execution.
- In other words, one thread can wait for another to complete using the `join()` method.
- Invoking `join()` guarantees that the method will not return until the threads started from it returns and join.

# Example – join()

In this demo we will see how a method works with and without join .

- Step 1: We will create a Runnable class **JoinEx** and a main class **JoinExMain**
- Step 2: First we will create the main method without join() and look at the output
- Step 3: Then will inject join() to the main method and see how join changes the output.

# Solution – join()

```
public class JoinEx implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Printing from "
                               + Thread.currentThread().getName()
                               + " , the value of i : " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Main method without join, create two threads and start it.

```
public class JoinExMain {  
    public static void main(String args[]) throws InterruptedException {  
        JoinEx r1 = new JoinEx();  
        JoinEx r2 = new JoinEx();  
        Thread t1 = new Thread(r1, "first");  
        Thread t2 = new Thread(r2, "second");  
        t1.start();  
        t2.start();  
        System.out.println("Main Thread Ends Here ");  
    }  
}
```



# Output – without join()

The main method ends before the completion of the other threads



Main Thread Ends Here

```
Printing from second , the value of i :0
Printing from first , the value of i :0
Printing from second , the value of i :1
Printing from first , the value of i :1
Printing from second , the value of i :2
Printing from first , the value of i :2
Printing from first , the value of i :3
Printing from second , the value of i :3
Printing from first , the value of i :4
Printing from second , the value of i :4
```

# Add the join()

```
public class JoinExMain {  
    public static void main(String args[]) throws InterruptedException {  
        JoinEx r1 = new JoinEx();  
        JoinEx r2 = new JoinEx();  
        Thread t1 = new Thread(r1, "first");  
        Thread t2 = new Thread(r2, "second");  
        t1.start();  
        t2.start();  
        t1.join();  
        t2.join();  
        System.out.println("Main Thread Ends Here ");  
    }  
}
```

The threads join method is triggered.

# Output – with join()

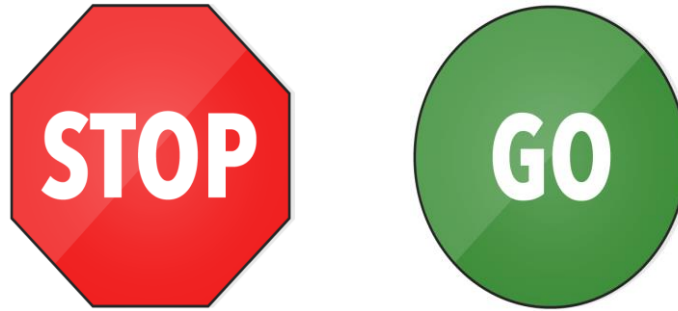
```
Printing from second , the value of i :0  
Printing from first , the value of i :0  
Printing from second , the value of i :1  
Printing from first , the value of i :1  
Printing from second , the value of i :2  
Printing from first , the value of i :2  
Printing from second , the value of i :3  
Printing from first , the value of i :3  
Printing from second , the value of i :4  
Printing from first , the value of i :4
```

Main Thread Ends Here



The main method ends after the completion of the other threads

# Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is Runnable interface?
- Which method is better either Thread or Runnable? Why?
- What is Thread join() method?

Thank you

*You have successfully completed*  
**Thread-II**

**JAVA @11**

Java Thread – Part III

# Objective

After completing this session you will be able to understand,

- What is synchronization?
- Various thread method wait, notify and notifyAll

# What is race condition?

**Scenario:** Consider a scenario in which you want to keep green blocks sequentially at the bottom and red blocks be placed on top of the green blocks. You have developed a program in such a way that two threads one for placing the green blocks and other for red blocks. Lets see how the program places the blocks.



You couldn't get your expected output. The red blocks and greens blocks were placed randomly. This is what we call a **Race Condition**



# When does race condition occur?

- A race condition occurs when two or more threads are able to access shared data and they try to change it at the same time.
- Both of them try to modify the data at the same time that is both are racing to get access/change the data
- Since the thread scheduling is system dependent we cannot predict a reliable behavior of the application.

**So what can be done to make only one Thread access a shared resource at a time ?**

**The solution is **synchronization**.**

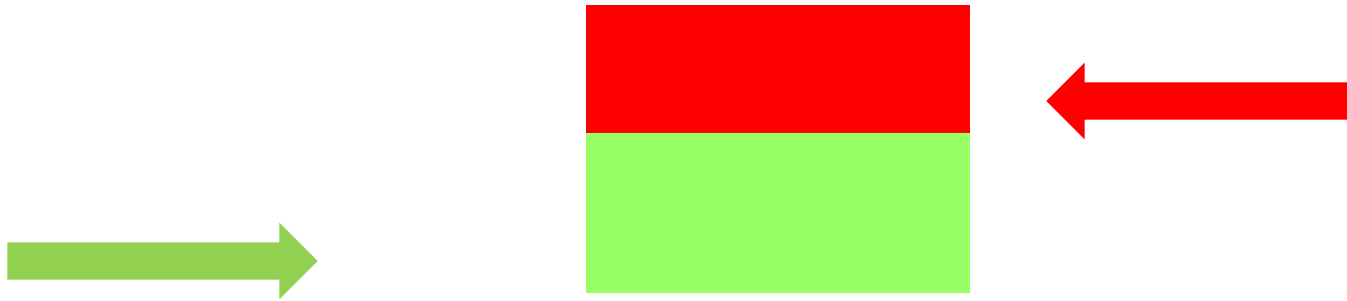
# What is synchronization?

***Synchronizing*** threads means access to shared data (or) method execution logic in a multithreaded application is controlled in such a way that only one thread can access the method/shared data at a time. The other threads will wait till the thread releases the control. The shared data is considered to be ***Thread Safe***.

Java uses the concept of ***monitors*** to implement synchronization.

# Problem solved using Synchronization?

Lets see how synchronization can solve our problem in placing the colored blocks



Here the red thread waits till the green thread completes the job.  
This is achieved using **Synchronization**.

# Thread Monitor

- A **monitor** is like a room of a building that can be occupied by only one thread at a time.
- The room contains a data or logic.
- From the time a thread enters this room to the time it leaves, it has exclusive access to any data/logic in the room.
  - This is done by gaining control on the monitor which is nothing but the room.
- Entering the special room inside the building is called "**acquiring the monitor**".
- Occupying the room is called "**owning the monitor**" or exclusive access to the data.
- Leaving the room is called "**releasing the monitor**".

# Synchronization methods

***Synchronization*** is achieved by using one of the two methods,

**Method 1:** Synchronized blocks

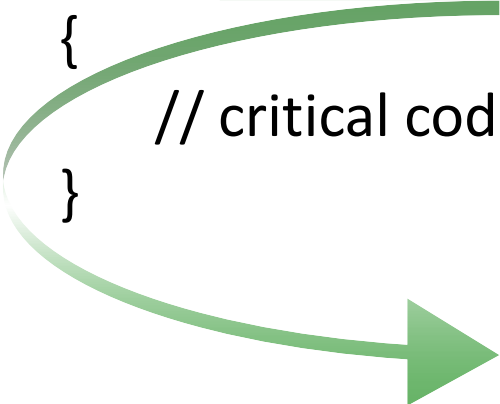
**Method 2:** Synchronized methods

# Method 1: synchronized method

- ***Synchronized methods*** can be created by using the keyword ***synchronized*** for defining the method.
- The code which should be thread safe is written inside this method.
- During the execution of a synchronized method, the thread holds the monitor of that method's object (or) if the method is static it holds the monitor of that method's class.
- If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

# Synchronizing a Method

```
public synchronized void updateRecord()  
{  
    // critical code goes here ...  
}
```



Only one thread will be inside the body of the `updateRecord()` method. The second call will be blocked until the first call returns or `wait()` is called inside the synchronized method.

# Synchronizing an Object

```
public void updateRecord()
{
    synchronized (this)
    {
        // critical code goes here ...
    }
}
```



# Example – Synchronize method

***Demonstration:*** In this demo we will see how to implement synchronization and how it works?. We will create a small application which prints a text message on the console.

This demo will also help you to understand the behavior of synchronized and unsynchronized methods.

# Example – Synchronize method

We will develop a program to spawn three threads, each thread should accept two string arguments and print the String message.

The following two messages should be printed by each thread,

Thread 1 – “Hello..” and “There”

Thread 2 – “How” and “are you”

Thread 3 – “Thank you,” and “very much”

## Classes

1. **PrinterThread** : Thread class used for printing the String.
2. **StringPrinter** : This class is used by the Thread class to print the Strings, contains a method which prints the strings.
3. **SyncExMain** : Main method to initiate the threads.

## Output Expected

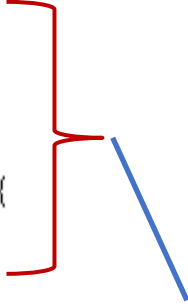
Hello..There

How are you

Thank you,very much

# Solution – Synchronize method

```
public class StringPrinter {  
    static void printStrings(String stringA, String stringB) {  
        System.out.print(stringA);  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException ie) {  
            ie.printStackTrace();  
        }  
        System.out.println(stringB);  
    }  
}
```



Methods to print the strings.

NOTE:

- The strings are printed one after the other with a pause. This has been implemented using the sleep method.
- The first message should be printed without a new line.
- Second message should be printed with a new line.


# Solution – Synchronize method

```
public class PrinterThread implements Runnable {  
    Thread t;  
    String stringA;  
    String stringB;  
  
    public PrinterThread(String stringA, String stringB) {  
        this.stringA = stringA;  
        this.stringB = stringB;  
        Thread t = new Thread(this);  
        t.start();  
    }  
  
    public void run() {  
        StringPrinter.printStrings(stringA, stringB);  
    }  
}
```

Creates a thread object on “this” Runnable object and starts the thread.

# Solution – Synchronize method

```
public class SyncExMain {  
    public static void main(String args[]) {  
        new PrinterThread("Hello..", "There");  
        new PrinterThread("How", " are you");  
        new PrinterThread("Thank you,", "very much");  
    }  
}
```



Create three threads and pass the appropriate String messages.

# Output – without sync method

Execute the class, the output could be displayed as below.

```
Thank you, HowHello..very much  
are you  
There
```

This is the output we get, not the expected one. Note the order in which the messages are printed could be different depending upon the OS scheduler.

Inference:

Each thread overruns the other thread resulting in the message being jumbled.

# add - synchronization

Now lets control the threads by using *synchronized* key word.

Make the *printStrings()* method of the *StringPrinter* class *synchronized*.

```
public class StringPrinter {  
    static synchronized void printStrings(String stringA, String stringB) {  
        System.out.print(stringA);  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException ie) {  
            ie.printStackTrace();  
        }  
        System.out.println(stringB);  
    }  
}
```

Add the *synchronized* keyword in method declaration.

# Output – with sync method

Execute the class, the output could be displayed as below.

Hello..There

How are you

Thank you,very much

**The output is as expected.**

Inference:

The *synchronized* keyword has ensured that the thread executes in order. Only after a thread completes printing both the messages the other thread starts the printing.



# Method 2: sync statements

## Where is Synchronized statement used?

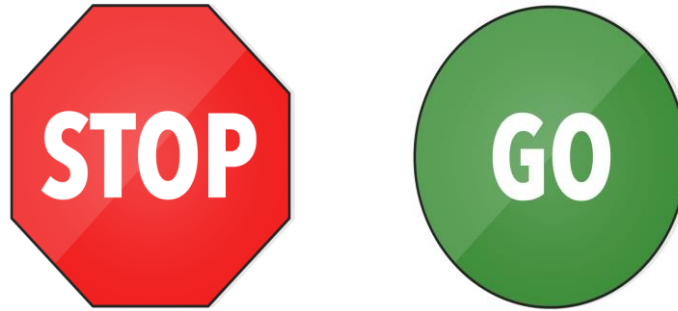
If only a small portion of code in a method needs to be Thread safe we can mark that portion only as synchronized rather than marking the entire method as ***synchronized***.

## Syntax:

```
public class SyncMethods{  
    public void methodA() {  
        synchronized(this){  
            //This block contains a set of  
            //statements which needs to be synchronized  
        }  
    }  
}
```

The syntax is **synchronized(this)** where **this** denotes the current object meaning the thread should acquire a lock on the current object to execute the statements.

# Time To Reflect



**Trainees to reflect the following topics before proceeding.**

- What is the need of synchronization in multi threaded programming?
- What is a race condition?
- How can Synchronization be implemented?

# Inter Thread Communication


In many instance we end up in a situation where we need one thread to communicate with other threads in a process. This is referred to as “***Inter Thread Communication***”

Inter Thread communication is achieved using one or more of the below methods,

1.wait()

2.notify()

3.notifyAll()



Occasionally used in  
application  
development.

# wait()

- ***wait()*** method causes a thread to release the lock it is holding on an object allowing another thread to run.
- ***wait()*** can only be invoked within a synchronized code.
- It should always be wrapped in a try block and handle ***IOException***.
- ***wait()*** can only be invoked by the thread that owns the lock on the object using **synchronized(this)**.

# How wait method works?

- When ***wait()*** is invoked, the thread becomes dormant until one of the below four things occur,
  - Another thread invokes the ***notify()*** method for this object and the scheduler arbitrarily chooses to run the thread.
  - Another thread invokes the ***notifyAll()*** method for this object.
  - Another thread interrupts this thread.
  - The specified ***wait()*** time elapses.
- When one of the above occurs, the thread becomes re-available to the Thread scheduler and competes for a lock on the object.
- Once it regains the lock on the object, everything resumes as if no suspension had occurred.

# notify()

- **notify()** method wakes up a single thread that is waiting on the monitor of this object
- If many threads are waiting on this object, then one of them is chosen to be awakened.
- The choice is arbitrary and occurs at the discretion of the JVM implementation.
- **notify()** method can only be used within synchronized code.
- The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.

# notifyAll()

- Similar to notify method **notifyAll()** is also used to wake up threads that wait on an object
- The difference is **notify()** awakens only one thread whereas **notifyAll()** awakens all the threads

# Deadlocks

**Deadlock** describes a situation where two or more threads are blocked forever, waiting for each other.

Example to understand dead locks:

Tim & Ron are two good friends they belong to a country where they have a weird culture, whenever they meet each other one has to bow and wish the other “*Good Day*” he has to remain bowed till the other person bows back and greets back “*Thank you*”

Lets see what happened?



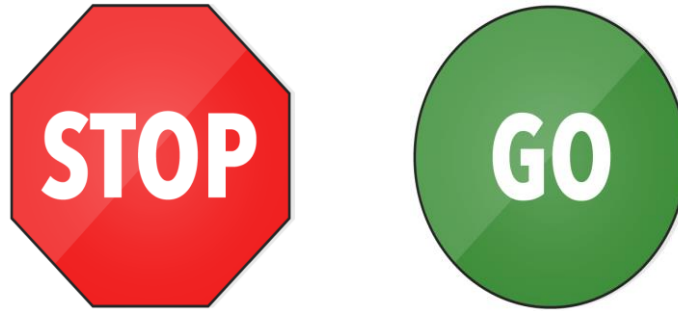
# Example – Inter thread communication

```
class Customer{
    int balance = 10000;
    synchronized void withdraw(int amount) {
        System.out.println("I am going to withdraw amount...");
        if(this.balance<amount) {
            System.out.println("Less balance; wait for some time.");
            try {
                wait();
            }catch(Exception e) {}
        }
        this.balance-=amount;
        System.out.println("Amount Received. Thank you.!");
    }
    synchronized void deposit(int amount) {
        System.out.println("Going to Deposit...");
        this.balance+=amount;
        System.out.println("Deposit Completed..");
        notify();
    }
}
```

# Example – Inter thread communication

```
public class InterThreadDemo {  
    public static void main(String[] args) {  
        final Customer cust = new Customer();  
        new Thread() {  
            public void run() {  
                cust.withDraw(9000);  
            }  
        }.start();  
        new Thread() {  
            public void run() {  
                cust.deposit(10000);  
            }  
        }.start();  
    }  
}
```

# Time To Reflect



**Trainees to reflect the following topics before proceeding.**

- How can we make a thread wait on an object it holds the lock?
- What is the difference between notify and notify all?
- What is a dead lock?

Thank you

*You have successfully completed*  
**Thread-III**