**Chapter 2 – Unit 3**

**Multithreading**

**The Java Thread model, thread life cycle, Thread class, Runnable interface, creating multiple threads, Synchronization, Inter Thread Communication, Deadlock.**

* **The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java.**
* **Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.
* A thread is a lightweight sub-process, the smallest unit of processing.
* Multiprocessing and multithreading, both are used to achieve multitasking.
* However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
* Java Multithreading is mostly used in games, animation, etc.
* Advantages of Java Multithreading
* 1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
* 2) You can perform many operations together, so it saves time.
* 3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

**Multitasking**

**Multitasking is a process of executing multiple tasks simultaneously**. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* **A process is heavyweight.**
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* **A thread is lightweight**.
* Cost of communication between the thread is low.

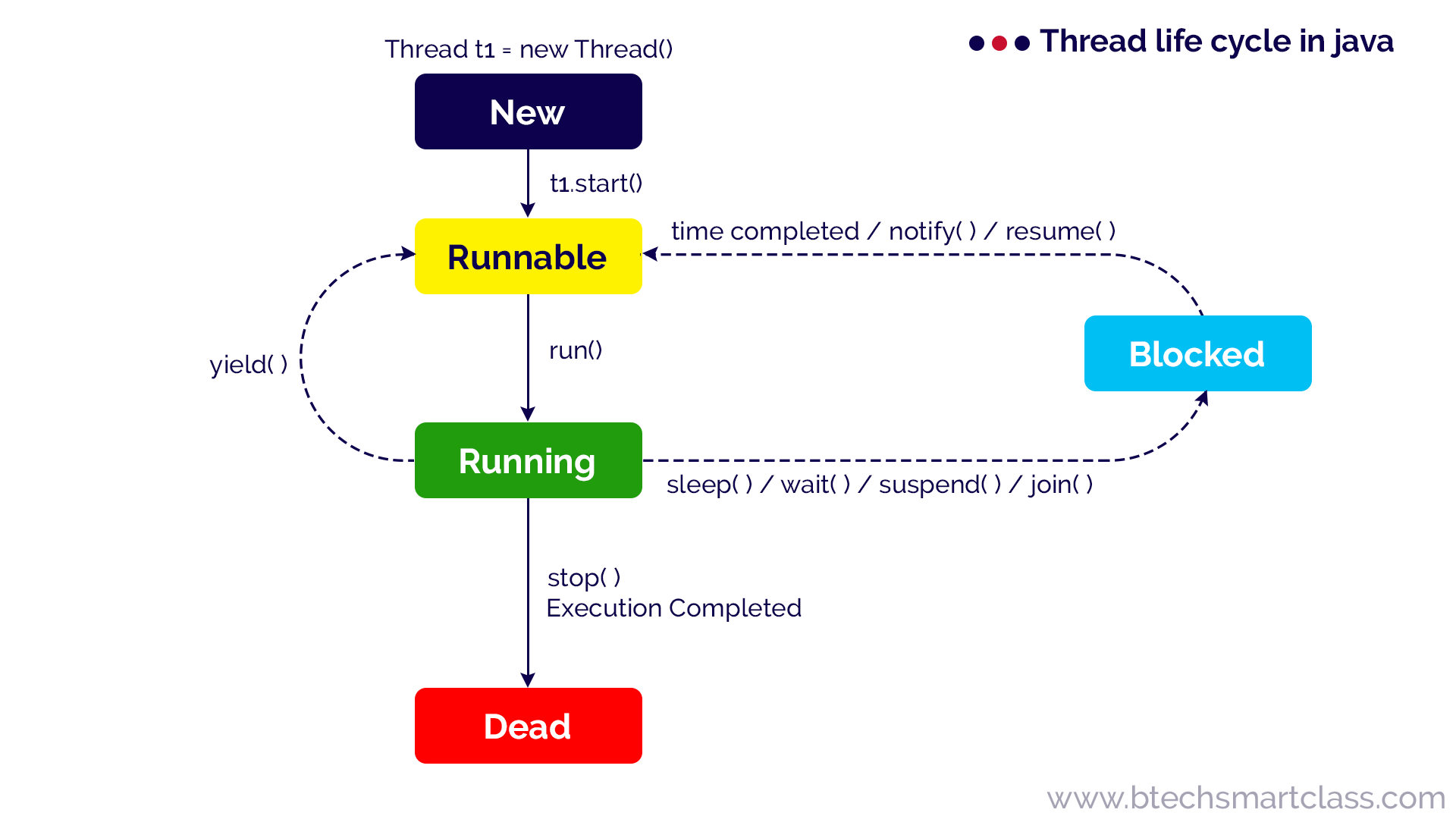
**What is Thread in java?**

* **A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.**
* Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



* As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), and one process can have multiple threads.

**Thread life cycle**



**New**

When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

**Runnable / Ready**

When a thread calls start( ) method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

**Running**

When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

**Blocked / Waiting**

A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

**Terminated or Dead:**

A thread terminates because of either of the following reasons:

* Because it exists normally. This happens when the code of the thread has been entirely executed by the program.
* Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

**The Main Thread**

* When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins.
* **The main thread is important for two reasons:**
* It is the thread from which other “child” threads will be spawned.
* Often, it must be the last thread to finish execution because it performs various shutdown actions.

**// Controlling the main Thread.**

**classCurrentThreadDemo**

**{**

**public static void main(String args[])**

**{**

**Thread t = Thread.currentThread();**

**System.out.println("Current thread: " + t);**

**// change the name of the thread**

**t.setName("My Thread");**

**System.out.println("After name change: " + t);**

**try**

**{**

**for(int n = 5; n > 0; n--)**

**{**

**System.out.println(n);**

**Thread.sleep(1000);**

**}**

**}**

**catch (InterruptedException e)**

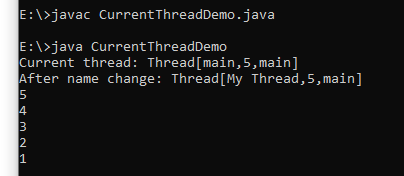
**{**

**System.out.println("Main thread interrupted");**

**}**

**}**

**}**

****

* **Notice the output produced when t is used as an argument to println( ).**
* **This displays, in order: the name of the thread, its priority, and the name of its group.**
* **By default, the name of the main thread is main. Its priority is 5, which is the default value, and main is also the name of the group of threads to which this thread belongs.**
* **A thread group is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, t is again output. This time, the new name of the thread is displayed.**

**Thread class**

* Java provides a thread class that has various method calls in order to manage the behavior of threads by providing constructors and methods to perform operations on threads.
* **Ways of creating threads**
* Creating own class which is extending to parent Thread class.
* Implementing the Runnable interface.

**Example : Implementing the Runnable interface**

// Create a second thread.

class NewThread implements Runnable

{

Thread t;

NewThread()

{

// Create a new, second thread

t = new Thread(this, "Demo Thread");

System.out.println("Child thread: " + t);

t.start(); // Start the thread

}

// This is the entry point for the second thread.

public void run()

{

try

{

for(inti = 5; i> 0; i--)

{

System.out.println("Child Thread: " + i);

Thread.sleep(500);

}

}

catch (InterruptedException e)

{

System.out.println("Child interrupted.");

}

System.out.println("Exiting child thread.");

}

}

classThreadDemo

{

public static void main(String args[])

{

new NewThread(); // create a new thread

//(or) NewThread t=new NewThread(); // create a new thread

try

{

for(inti = 5; i> 0; i--)

{

System.out.println("Main Thread: " + i);

Thread.sleep(1000);

}

}

catch (InterruptedException e)

{

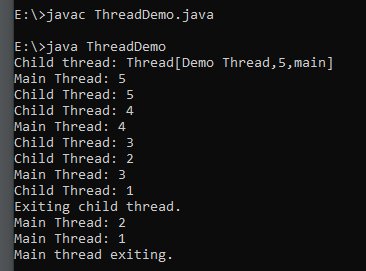
System.out.println("Main thread interrupted.");

}

System.out.println("Main thread exiting.");

}

}



Note :

* After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.
* Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName)

* In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin.
* The name of the new thread is specified by threadName.

**Example : Creating own class which is extending to parent Thread class.**

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread. Here is the preceding program rewritten to extend Thread:

// Create a second thread by extending Thread

class NewThread extends Thread

{

NewThread()

{

// Create a new, second thread

super("Demo Thread");

System.out.println("Child thread: " + this);

start(); // Start the thread

}

// This is the entry point for the second thread.

public void run()

{

try

{

for(int i = 5; i > 0; i--)

{

System.out.println("Child Thread: " + i);

Thread.sleep(500);

}

}

catch (InterruptedException e)

{

System.out.println("Child interrupted.");

}

System.out.println("Exiting child thread.");

}

}

class ExtendThread

{

public static void main(String args[])

{

//new NewThread(); // create a new thread

NewThread t=new NewThread();

try

{

for(int i = 5; i > 0; i--)

{

System.out.println("Main Thread: " + i);

Thread.sleep(1000);

}

}

catch (InterruptedException e)

{

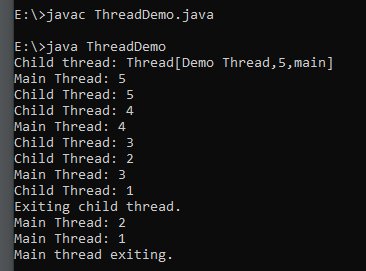
System.out.println("Main thread interrupted.");

}

System.out.println("Main thread exiting.");

}

}



Note :

* This program generates the same output as the preceding version.
* As you can see, the child thread is created by instantiating an object of NewThread, which is derived from Thread.
* Notice the call to super( ) inside NewThread.
* This invokes the following form of the Thread constructor:

public Thread(String threadName)

* Here, threadName specifies the name of the thread.

**Creating Multiple**

* Threads So far, you have been using only two threads: the main thread and one child thread.
* However, your program can spawn as many threads as it needs.
* For example, the following program creates three child threads:

// Create multiple threads.

class NewThread implements Runnable

{

String name; // name of thread

Thread t;

NewThread(String threadname)

{

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

t.start(); // Start the thread

}

// This is the entry point for thread.

public void run()

{

try {

for(int i = 5; i > 0; i--)

{

System.out.println(name + ": " + i);

Thread.sleep(1000);

}

}

catch (InterruptedException e)

{

System.out.println(name + "Interrupted");

}

System.out.println(name + " exiting.");

}

}

class MultiThreadDemo

{

public static void main(String args[])

{

//new NewThread("One"); // start threads

//new NewThread("Two");

//new NewThread("Three");

NewThread one =new NewThread("One"); // start threads

NewThread two =new NewThread("Two");

NewThread three=new NewThread("Three");

try

{

// wait for other threads to end

Thread.sleep(10000);

}

catch (InterruptedException e)

{

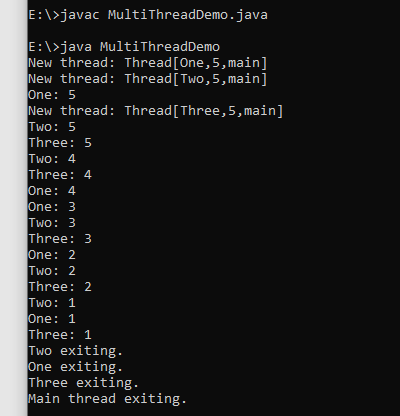
System.out.println("Main thread Interrupted");

}

System.out.println("Main thread exiting.");

}

}



**isAlive() method**

* The isAlive() method of thread class tests if the thread is alive.
* A thread is considered alive when the start() method of thread class has been called and the thread is not yet dead.
* This method returns true if the thread is still running and not finished.
* **This method will return true if the thread is alive otherwise returns false.**

**Syntax**

public final boolean isAlive()

**join() method**

* The join() method of thread class waits for a thread to die.
* **It is used when you want one thread to wait for completion of another.**
* This process is like a relay race where the second runner waits until the first runner comes and hand over the flag to him.

**Syntax**

public final void join()throws InterruptedException

public void join(long millis)throwsInterruptedException

public final void join(long millis, int nanos)throws InterruptedException

**Parameter**

1. millis: It defines the time to wait in milliseconds.
2. nanos: 0-999999 additional nanoseconds to wait.

// Using join() to wait for threads to finish.

class NewThread implements Runnable

{

String name; // name of thread

Thread t;

NewThread(String threadname)

{

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

t.start(); // Start the thread

}

// This is the entry point for thread.

public void run()

{

try

{

for(int i = 5; i > 0; i--)

{

System.out.println(name + ": " + i);

Thread.sleep(1000);

}

}

catch (InterruptedException e)

{

System.out.println(name + " interrupted.");

}

System.out.println(name + " exiting.");

}

}

class DemoJoin

{

public static void main(String args[])

{

NewThread ob1 = new NewThread("One");

NewThread ob2 = new NewThread("Two");

NewThread ob3 = new NewThread("Three");

System.out.println("Thread One is alive: "+ ob1.t.isAlive());

System.out.println("Thread Two is alive: "+ ob2.t.isAlive());

System.out.println("Thread Three is alive: "+ ob3.t.isAlive());

// wait for threads to finish

try

{

System.out.println("Waiting for threads to finish.");

ob1.t.join();

ob2.t.join();

ob3.t.join();

}

catch (InterruptedException e)

{

System.out.println("Main thread Interrupted");

}

System.out.println("Thread One is alive: "+ ob1.t.isAlive());

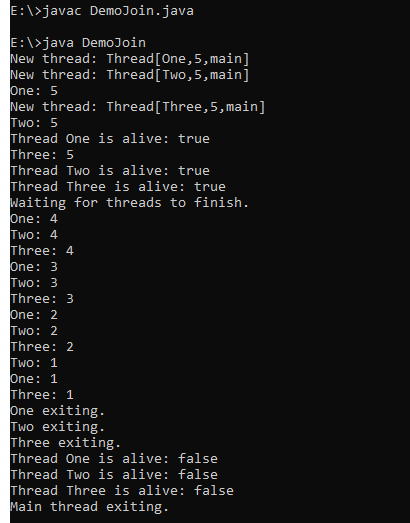
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());

System.out.println("Thread Three is alive: "+ ob3.t.isAlive());

System.out.println("Main thread exiting.");

}

}



**Thread Priorities**

* Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads.
* To set a thread’s priority, use the setPriority( ) method, which is a member of Thread.
* This is its general form:

final void setPriority(int level)

* Here, level specifies the new priority setting for the calling thread.
* **The value of level must be within the range MIN\_PRIORITY and MAX\_PRIORITY. Currently, these values are 1 and 10, respectively.**
* **To return a thread to default priority, specify NORM\_PRIORITY, which is currently 5.**
* These priorities are defined as **static final variables** within Thread.
* You can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

final int getPriority( )

**Synchronization**

### Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

**TestSynchronization1.java**

class Table

{

void printTable(int n)

{

//method not synchronized

for(int i=1;i<=5;i++)

{

System.out.println(n\*i);

Try

{

Thread.sleep(400);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

}

class MyThread1 extends Thread

{

Table t;

MyThread1(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(5);

}

}

class MyThread2 extends Thread

{

Table t;

MyThread2(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(100);

}

}

class TestSynchronization1

{

public static void main(String args[])

{

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

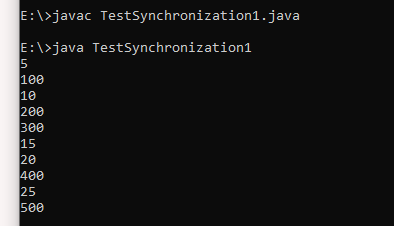
MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}

****

**The synchronized Statement:**

* If you declare any method as synchronized, it is known as synchronized method.
* Synchronized method is used to lock an object for any shared resource.
* **When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.**
* **It can be achieved by using the following three ways:**
* By Using Synchronized Method
* By Using Synchronized Block
* By Using Static Synchronization

**//example of java synchronized method**

**class Table**

**{**

**synchronized void printTable(int n)**

**{**

**//synchronized method**

**for(int i=1;i<=5;i++)**

**{**

**System.out.println(n\*i);**

**Try**

**{**

**Thread.sleep(400);**

**}**

**catch(Exception e){System.out.println(e);**

**}**

**}**

**}**

**}**

**class MyThread1 extends Thread**

**{**

**Table t;**

**MyThread1(Table t)**

**{**

**this.t=t;**

**}**

**public void run()**

**{**

**t.printTable(5);**

**}**

**}**

**class MyThread2 extends Thread**

**{**

**Table t;**

**MyThread2(Table t)**

**{**

**this.t=t;**

**}**

**public void run()**

**{**

**t.printTable(100);**

**}**

**}**

**public class TestSynchronization2**

**{**

**public static void main(String args[])**

**{**

**Table obj = new Table();//only one object**

**MyThread1 t1=new MyThread1(obj);**

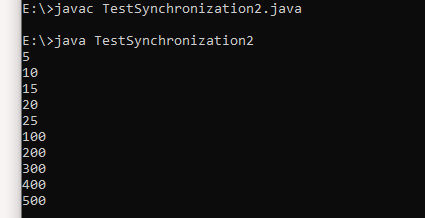
**MyThread2 t2=new MyThread2(obj);**

**t1.start();**

**t2.start();**

**}**

**}**

****

**Synchronized block**

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to Remember

* Synchronized block is used to lock an object for any shared resource.
* Scope of synchronized block is smaller than the method.
* A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
* The system performance may degrade because of the slower working of synchronized keyword.
* Java synchronized block is more efficient than Java synchronized method.

Syntax

* This is the general form of the synchronized statement:

synchronized(object)

{

// statements to be synchronized

}

Here, object is a reference to the object being synchronized.

A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object’s monitor.

**class Table**

**{**

**void printTable(int n)**

**{**

**synchronized(this)**

**{**

**//synchronized block**

**for(int i=1;i<=5;i++)**

**{**

**System.out.println(n\*i);**

**Try**

**{**

**Thread.sleep(400);**

**}**

**catch(Exception e)**

**{**

**System.out.println(e);**

**}**

**}**

**}**

**}**

**//end of the method**

**}**

**class MyThread1 extends Thread**

**{**

**Table t;**

**MyThread1(Table t)**

**{**

**this.t=t;**

**}**

**public void run()**

**{**

**t.printTable(5);**

**}**

**}**

**class MyThread2 extends Thread**

**{**

**Table t;**

**MyThread2(Table t)**

**{**

**this.t=t;**

**}**

**public void run()**

**{**

**t.printTable(100);**

**}**

**}**

**class TestSynchronizedBlock1**

**{**

**public static void main(String args[])**

**{**

**Table obj = new Table();//only one object**

**MyThread1 t1=new MyThread1(obj);**

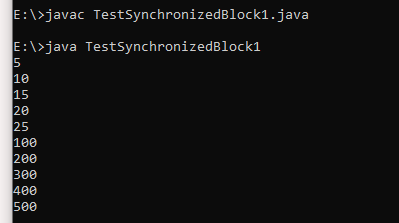
**MyThread2 t2=new MyThread2(obj);**

**t1.start();**

**t2.start();**

**}**

**}**

****

**Inter Thread Communication**

**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**:

* wait()
* notify()
* notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

* These methods are declared within Object, as shown here:

final void wait( ) throws InterruptedException

final void notify( )

final void notifyAll( )

**Example**

**class Customer**

**{**

**int amount=10000;**

**synchronized void withdraw(int amount)**

**{**

**System.out.println("going to withdraw...");**

**if(this.amount<amount)**

**{**

**System.out.println("Less balance; waiting for deposit...");**

**System.out.println("Before deposite - current balance:"+this.amount);**

**try**

**{**

**wait();**

**}**

**catch(Exception e)**

**{**

**}**

**}**

**this.amount-=amount;**

**System.out.println("withdraw completed...");**

**System.out.println("After withdraw - current balance:"+this.amount);**

**}**

**synchronized void deposit(int amount)**

**{**

**System.out.println("going to deposit...");**

**this.amount+=amount;**

**System.out.println("deposit completed... ");**

**System.out.println("After deposite - current balance:"+this.amount);**

**notify();**

**}**

**}**

**class Test**

**{**

**public static void main(String args[])**

**{**

**final Customer c=new Customer();**

**new Thread()**

**{**

**public void run()**

**{**

**c.withdraw(15000);**

**}**

**}.start();**

**new Thread()**

**{**

**public void run()**

**{**

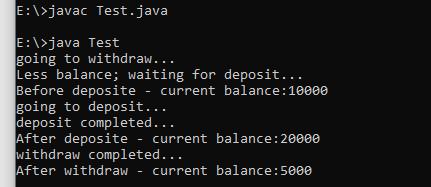
**c.deposit(10000);**

**}**

**}.start();**

**}**

**}**

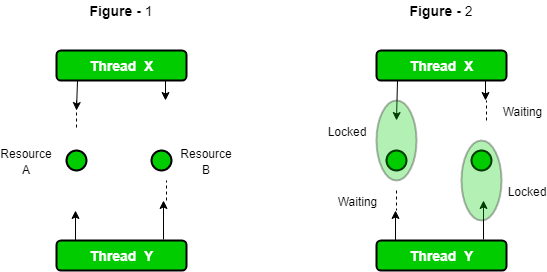
****

**Deadlock**

* Deadlock in Java is a part of multithreading.
* **Deadlock can occur in a situation when a thread1 is waiting for an object lock, that is acquired by thread2 and thread2 is waiting for an object lock that is acquired by thread1.**
* Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



* [Synchronized](https://www.geeksforgeeks.org/synchronized-in-java/) keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.
* It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called [Deadlock](https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/).
* Below is a simple example of Deadlock condition.



class TestDeadlockExample1

{

public static void main(String[] args)

{

final String resource1 = "RVR";

final String resource2 = "CSE";

// t1 tries to lock resource1 then resource2

Thread t1 = new Thread()

{

public void run()

{

synchronized (resource1)

{

System.out.println("Thread 1: locked resource 1");

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

{

}

synchronized (resource2)

{

System.out.println("Thread 1: locked resource 2");

}

}

}

};

// t2 tries to lock resource2 then resource1

Thread t2 = new Thread()

{

public void run()

{

synchronized (resource2)

{

System.out.println("Thread 2: locked resource 2");

try

{

Thread.sleep(100);

}

catch (Exception e)

{

}

synchronized (resource1)

{

System.out.println("Thread 2: locked resource 1");

}

}

}

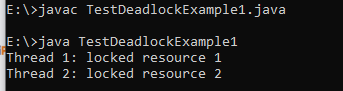
};

t1.start();

t2.start();

}

}



### More Complicated Deadlocks

A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

### How to avoid deadlock?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

class DeadlockSolved

{

public static void main(String ar[])

{

DeadlockSolved test = new DeadlockSolved();

final resource1 a = test.new resource1();

final resource2 b = test.new resource2();

// Thread-1

Runnable b1 = new Runnable()

{

public void run()

{

synchronized (b)

{

try

{

/\* Adding delay so that both threads can start trying to lock resources \*/

Thread.sleep(100);

}

catch (InterruptedException e)

{

e.printStackTrace();

}

// Thread-1 have resource1 but need resource2 also

synchronized (a)

{

System.out.println("In block 1");

}

}

}

};

// Thread-2

Runnable b2 = new Runnable()

{

public void run()

{

synchronized (b)

{

// Thread-2 have resource2 but need resource1 also

synchronized (a)

{

System.out.println("In block 2");

}

}

}

};

new Thread(b1).start();

new Thread(b2).start();

}

// resource1

private class resource1

{

private int i = 10;

public int getI()

{

return i;

}

public void setI(int i)

{

this.i = i;

}

}

// resource2

private class resource2

{

private int i = 20;

public int getI()

{

return i;

}

public void setI(int i)

{

this.i = i;

}

}

}

