# Multithreaded Programming

Vaidehi Patel
Computer Engineering
Indus University

## Content

- Use of Multithread programming,
- Thread class and Runnable interface,
- Thread priority,
- Thread synchronization,
- Thread communication,
- Deadlock.

## Introduction

- **Multithreading in Java** 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.

- 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)
- **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 memory maps, updating lists, etc.
- **Thread-based Multitasking (Multithreading)**
  - Threads share the same address space.
  - A thread is lightweight.
  - Cost of communication between the thread is low.

# Java Thread class

- Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

| S.N. | Modifier and Type | Method | Description |
|---|---|---|---|
| 1) | void | start() | It is used to start the execution of the thread. |
| 2) | void | run() | It is used to do an action for a thread. |
| 3) | static void | sleep() | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | currentThread() | It returns a reference to the currently executing thread object. |
| 5) | void | join() | It waits for a thread to die. |
| 6) | int | getPriority() | It returns the priority of the thread. |

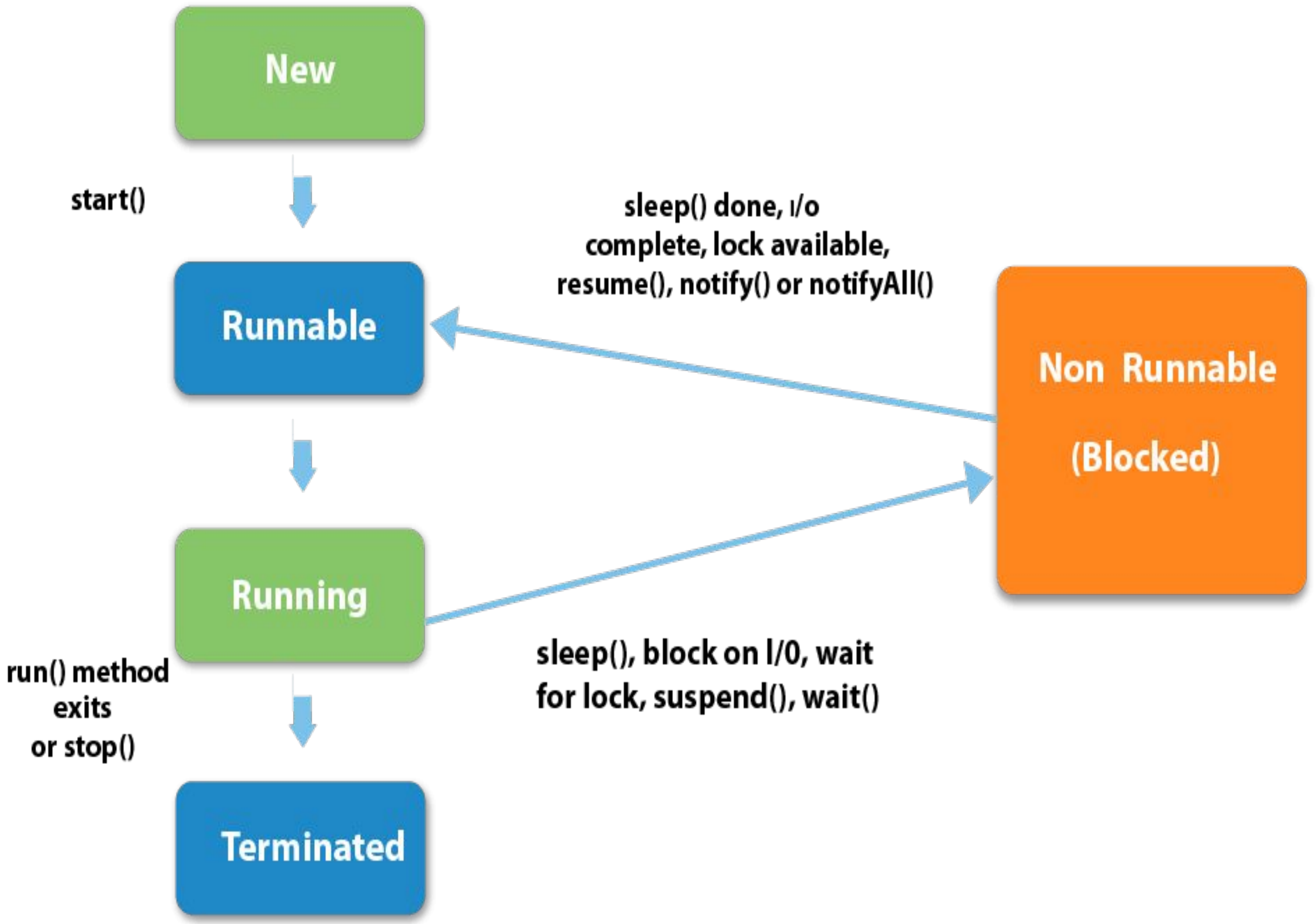
|  |  |  |  |
|---|---|---|---|
| 7) | void | setPriority() | It changes the priority of the thread. |
| 8) | String | getName() | It returns the name of the thread. |
| 9) | void | setName() | It changes the name of the thread. |
| 10) | long | getId() | It returns the id of the thread. |
| 11) | boolean | isAlive() | It tests if the thread is alive. |
| 12) | static void | yield() | It causes the currently executing thread object to pause and allow other threads to execute temporarily. |
| 13) | void | suspend() | It is used to suspend the thread. |
| 14) | void | resume() | It is used to resume the suspended thread. |
| 15) | void | stop() | It is used to stop the thread. |
| 16) | void | destroy() | It is used to destroy the thread group and all of its subgroups. |

| | | | |
|---|---|---|---|
| 17) | boolean | isDaemon() | It tests if the thread is a daemon thread. |
| 18) | void | setDaemon() | It marks the thread as daemon or user thread. |
| 19) | void | interrupt() | It interrupts the thread. |
| 20) | boolean | isinterrupted() | It tests whether the thread has been interrupted. |
| 21) | static boolean | interrupted() | It tests whether the current thread has been interrupted. |
| 22) | static int | activeCount() | It returns the number of active threads in the current thread's thread group. |
| 23) | void | checkAccess() | It determines if the currently running thread has permission to modify the thread. |
| 24) | static boolean | holdLock() | It returns true if and only if the current thread holds the monitor lock on the specified object. |
| 25) | static void | dumpStack() | It is used to print a stack trace of the current thread to the standard error stream. |
| 26) | StackTraceElement[ ] | getStackTrace() | It returns an array of stack trace elements representing the stack dump of the thread. |
| 27) | static int | enumerate() | It is used to copy every active thread's thread group and its subgroup into the specified array. |

| | | | |
|---|---|---|---|
| 28) | Thread.State | getState() | It is used to return the state of the thread. |
| 29) | ThreadGroup | getThreadGroup() | It is used to return the thread group to which this thread belongs |
| 30) | String | toString() | It is used to return a string representation of this thread, including the thread's name, priority, and thread group. |
| 31) | void | notify() | It is used to give the notification for only one thread which is waiting for a particular object. |
| 32) | void | notifyAll() | It is used to give the notification to all waiting threads of a particular object. |
| 33) | void | setContextClassLoader() | It sets the context ClassLoader for the Thread. |
| 34) | ClassLoader | getContextClassLoader() | It returns the context ClassLoader for the thread. |
| 35) | static Thread.UncaughtExceptionHandler | getDefaultUncaughtExceptionHandler() | It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception. |
| 36) | static void | setDefaultUncaughtExceptionHandler() | It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception. |

## Life cycle of a Thread (Thread States)

- A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
- But for better understanding the threads, we are explaining it in the 5 states.
- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:


New
start()
Runnable
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable (Blocked)
Running
sleep(), block on I/0, wait for lock, suspend(), wait()
run() method exits or stop()
Terminated

1) New
   - The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable
   - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running
   - The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)
   - This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated
   - A thread is in terminated or dead state when its run() method exits.

- How to create thread
- There are two ways to create a thread:
  - By extending Thread class
  - By implementing Runnable interface.
- Thread class:
  - Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.Commonly used Constructors of Thread class:
    - Thread()
    - Thread(String name)
    - Thread(Runnable r)
    - Thread(Runnable r,String name)

- Runnable interface:
- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().
  - **public void run():** is used to perform action for a thread.
- Starting a thread:
  - **start() method** of Thread class is used to start a newly created thread. It performs following tasks:A new thread starts(with new callstack).
  - The thread moves from New state to the Runnable state.
  - When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

- Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

## Thread Scheduler in Java

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process.
- Difference between preemptive scheduling and time slicing
- Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

## Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.
- Syntax of sleep() method in java
- The Thread class provides two methods for sleeping a thread:
- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.p
     rintln(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

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

# Priority of a Thread (Thread Priority):

- Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.3 constants defined in Thread class:
  - public static int MIN_PRIORITY
  - public static int NORM_PRIORITY
  - public static int MAX_PRIORITY
- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getN
     ame());
   System.out.println("running thread priority is:"+Thread.currentThread().get
     Priority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

# Thread Synchronization

- Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.
- Why use Synchronization
- The synchronization is mainly used to
  - To prevent thread interference.
  - To prevent consistency problem.
- Types of Synchronization
  - Process Synchronization
  - Thread Synchronization

- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive
  - Synchronized method.
  - Synchronized block.
  - static synchronization.
- Cooperation (Inter-thread communication in java)
- **Mutual Exclusive**
- Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:
  - by synchronized method
  - by synchronized block
  - by static synchronization

- Concept of Lock in Java
- Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
- From Java 5 the package java.util.concurrent.locks contains several lock implementations.

- Understanding the problem without Synchronization

```
class Table{
void printTable(int n){//method not sync
hronized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.printl
n(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();
}
}
```

```
Output: 5 100 10 200 15 300 20 400 25 500
```

- Java synchronized method
- 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.

```
class Table{
 synchronized void printTable(int n){//synchron
      ized 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();
}
}
```

Output: 5 10 15 20 25 100 200 300 400 500

# Synchronized Block in Java

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.
- **Points to remember for Synchronized block**
  - Synchronized block is used to lock an object for any shared resource.
  - Scope of synchronized block is smaller than the method.
- **Syntax to use synchronized block**

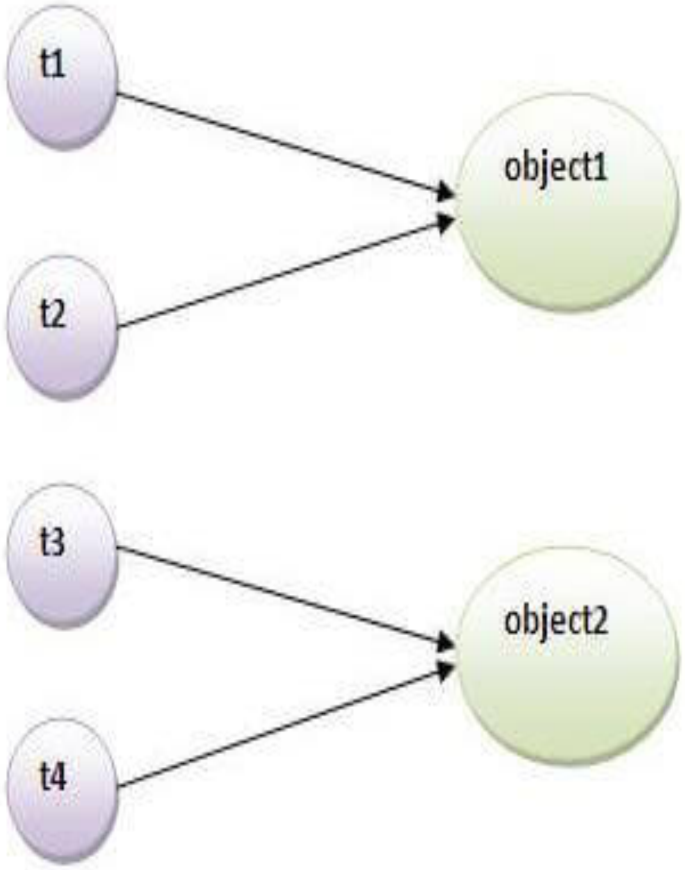
```
synchronized (object reference expression) {
  //code block
}
```

```
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);
}
}

public 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();
}
}
```

Output:5 10 15 20 25 100 200 300 400 500

# Static Synchronization

- If you make any static method as synchronized, the lock will be on the class not on object.
- Problem without static synchronization
- Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.
- But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.
- I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

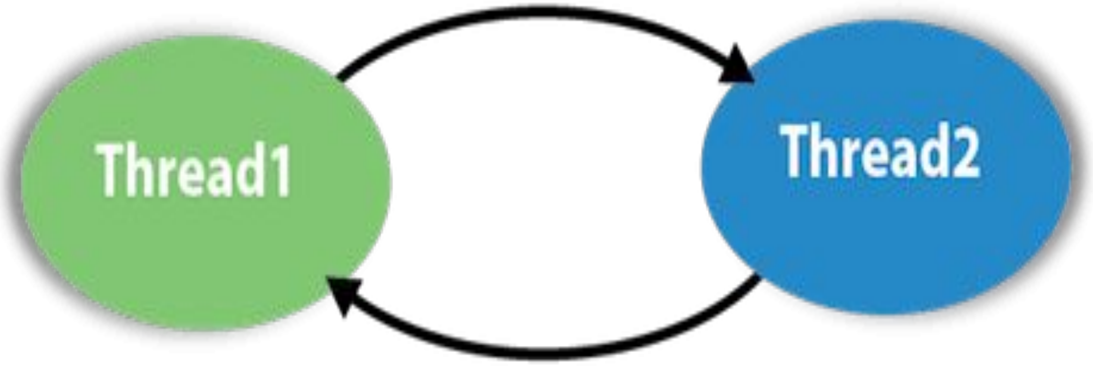

t1
t2
object1
t3
t4
object2

```
class Table{

 synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}

class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}

class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
```

```
class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
 class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}

public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

# Deadlock in java

- Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another 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.



```
public class TestDeadlockExample1 {
 public static void main(String[] args) {
  final String resource1 = "ratan jaiswal";
  final String resource2 = "vimal jaiswal";
  // t1 tries to lock resource1 then resource2

  Thread t1 = new Thread() {
   public void run() {
     synchronized (resource1) {
      System.out.println("Thread 1: locked re
    source 1");

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

      synchronized (resource2) {
       System.out.println("Thread 1: locked r
    esource 2");
      }
     }
   }
  };
// t2 tries to lock resource2 then resource1
  Thread t2 = new Thread() {
   public void run() {
    synchronized (resource2) {
     System.out.println("Thread 2: locked reso
urce 2");

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

     synchronized (resource1) {
      System.out.println("Thread 2: locked res
ource 1");
     }
    }
   }
  };


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

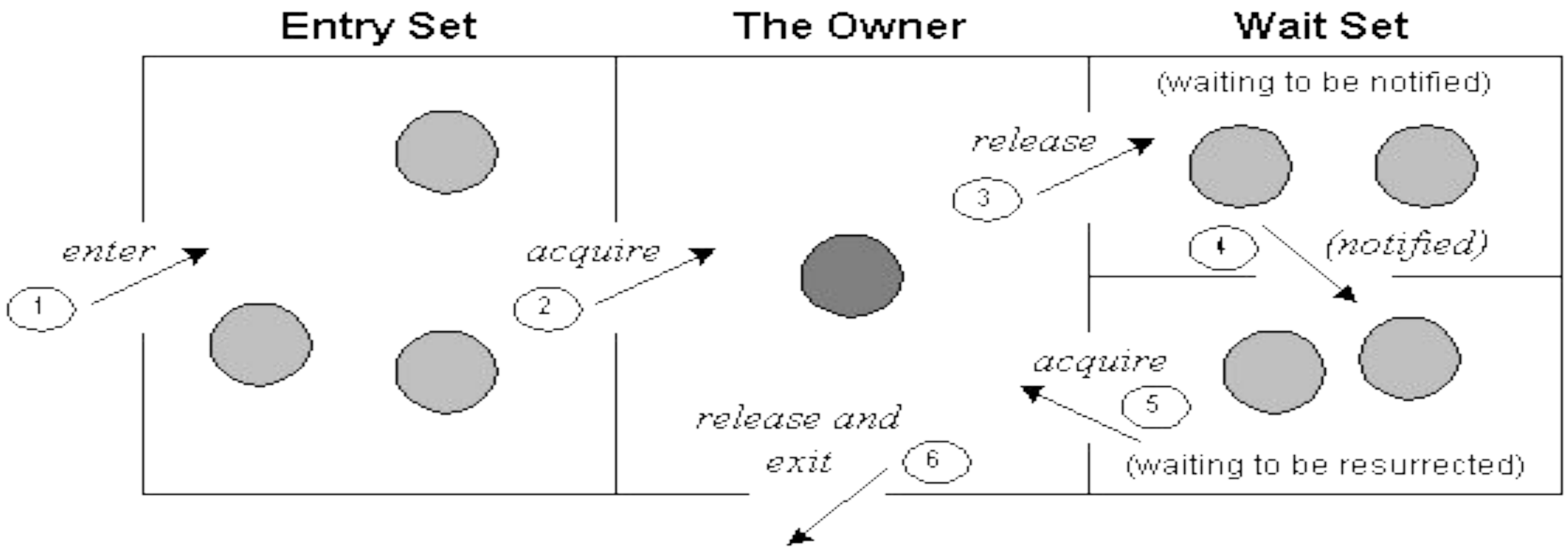
# Inter-thread communication in Java

- **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()
-

- 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.

| Method | Description |
|---|---|
| public final void wait()throws InterruptedException | waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | waits for the specified amount of time. |

- 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. Syntax:
  - public final void notify()
- notifyAll() method
  - Wakes up all threads that are waiting on this object's monitor. Syntax:
  - public final void notifyAll()



- Threads enter to acquire lock.
- Lock is acquired by on thread.
- Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
- If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
- Now thread is available to acquire lock.
- After completion of the task, thread releases the lock and exits the monitor state of the object.

# Difference between wait and sleep

| wait() | sleep() |
|---|---|
| wait() method releases the lock | sleep() method doesn't release the lock. |
| is the method of Object class | is the method of Thread class |
| is the non-static method | is the static method |
| is the non-static method | is the static method |
| should be notified by notify() or notifyAll() methods | after the specified amount of time, sleep is completed. |

```
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...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
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();

}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
 going to deposit...
deposit completed...
 withdraw completed
```