VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

CSE1007
Java Programming

Slot: A2+TA2
Module -III

Venue: AB1-409
Robustness and Concurrency

Prof. Tulasi Prasad Sariki
SCSE, VIT, Chennai
www.learnersdesk.weebly.com

Course Contents:

- Exception Handling
  - Exceptions & Errors
  - Types of Exception
  - Use of try, catch, finally, throw, throws
  - Control Flow in Exceptions
  - user defined exceptions
- Multithreading
  - Thread creation
  - sharing the workload among threads
  - Synchronization
  - Deadlock
  - Inter thread communication

Exception Handling

- Exception handling is one of the most important feature of java programming that allows us to handle the run-time errors caused by exceptions.
- What is an exception?

  An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

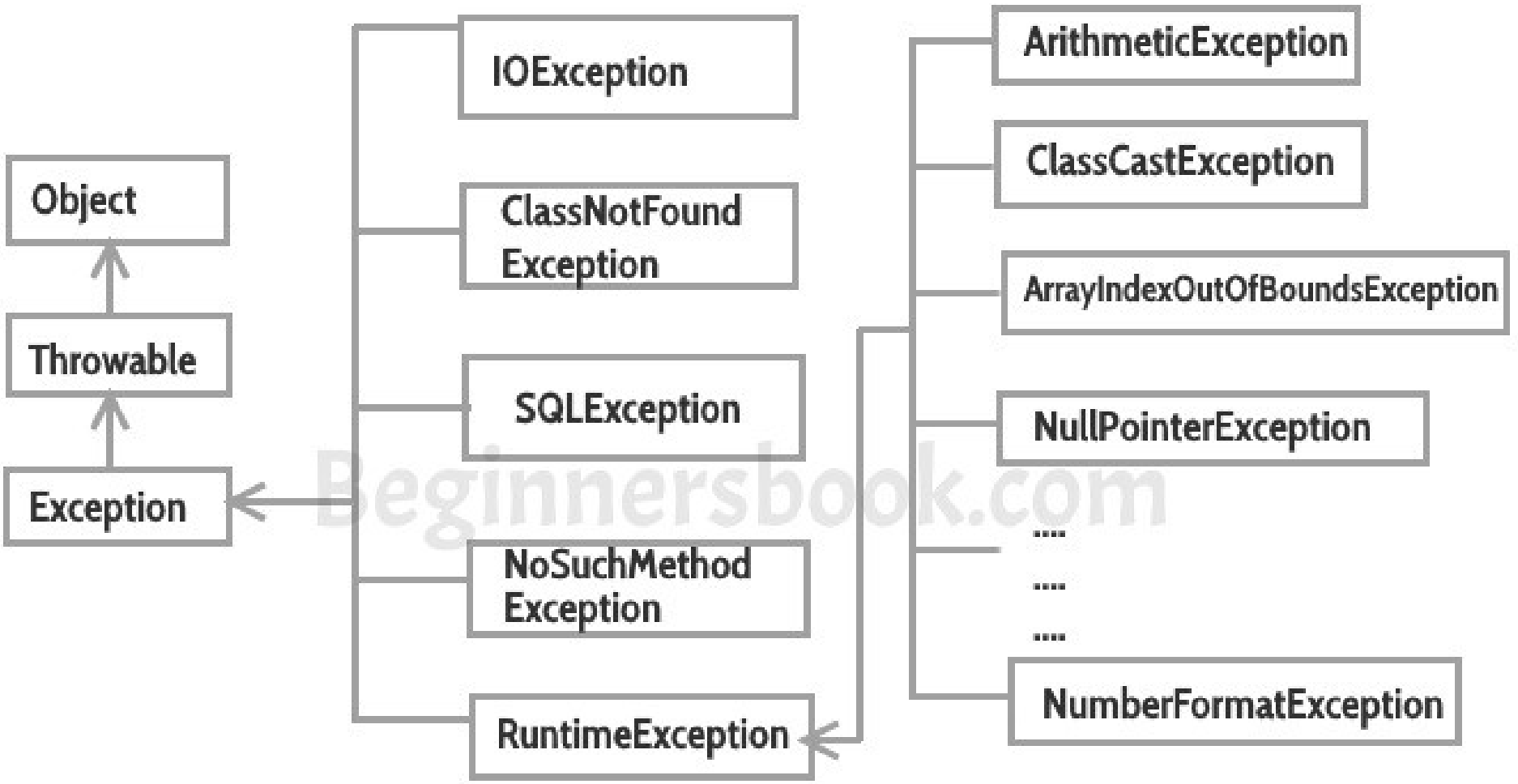
Exception Handling

- **Why an exception occurs?**
  - There can be several reasons that can cause a program to throw exception.
  - Example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.
- **Advantage of exception handling**
  - Exception handling ensures that the flow of the program doesn't break when an exception occurs.
  - By handling we make sure that all the statements execute and the flow of program doesn't break.

Error and Exception

- **Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.**
- **Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:**
  - NullPointerException – When you try to use a reference that points to null.
  - ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.
  - ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds,

Class Hierarchy of Exception



Types of exceptions

- **There are two types of exceptions in Java**
- **1. Checked exceptions:**
  - All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not.
  - If these exceptions are not handled/declared in the program, you will get compilation error.
  - Example, SQLException, IOException, ClassNotFoundException etc.

Types of exceptions

- **There are two types of exceptions in Java**
- **2. Unchecked exceptions**
  - Runtime Exceptions are also known as Unchecked Exceptions.
  - These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit.
  - For Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Try Block

- **Try block**
  - The try block contains set of statements where an exception can occur.
  - A try block is always followed by a catch block, which handles the exception that occurs in associated try block.
  - A try block must be followed by catch blocks or finally block or both
- **Syntax of try block**

```
try{
  //statements that may cause an exception
}
```

- **While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception**

Catch Block

- A catch block is where you handle the exceptions, this block must follow the try block.
- A single try block can have several catch blocks associated with it.
- You can catch different exceptions in different catch blocks.
- When an exception occurs in try block, the corresponding catch block that handles that particular exception executes.
- For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Try catch syntax

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

Example

```
class Example{
  public static void main(String args[]) {
    int num1, num2;
    try {
      Num1 = 0;  num2 = 62 / num1;
      System.out.println(num2);
      System.out.println("Hey I'm at the end of try block");
    }
    catch (ArithmeticException e) {
System.out.println("You should not divide a number by zero");}
        System.out.println("I'm out of try-catch block in Java.");   } }
```

Multiple catch blocks

- 1. A single try block can have any number of catch blocks.
- 2. A generic catch block can handle all the exceptions.
- 3. If no exception occurs in try block then the catch blocks are completely ignored.
- 4. Corresponding catch blocks execute for that specific type of exception:
  - catch(ArithmeticException e) is a catch block that can hanlde ArithmeticException
  - catch(NullPointerException e) is a catch block that can handle NullPointerException.
- Use can throw exception (User-defined exception)

Example

```
class Example{
  public static void main(String args[]){
    try{
       int a[]=new int[7];
       a[4]=30/0;
       System.out.println("First print statement in try block");
    }
    catch(ArithmeticException e){ System.out.println("Warning: ArithmeticException"); }
    catch(ArrayIndexOutOfBoundsException e){ System.out.println("Warning:
ArrayIndexOutOfBoundsException");}
    catch(Exception e){ System.out.println("Warning: Some Other exception");}
  System.out.println("Out of try-catch block...");}}
```

Finally block

- A finally block contains all the crucial statements that must be executed whether exception occurs or not.
- The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

try { //Statements that may cause an exception }

catch { //Handling exception }

finally { //Statements to be executed }

Example

```
class Example
{
   public static void main(String args[]) {
      try{
 int num=121/0;
 System.out.println(num);  }
catch(ArithmeticException e){  System.out.println("Number should not be divided by zero"); }
      finally{ System.out.println("This is finally block"); }
      System.out.println("Out of try-catch-finally");  } }
```

Finally block

- A finally block must be associated with a try block, you cannot use finally without a try block.
- You should place those statements in this block that must be executed always.
- Finally block is optional, a try-catch block is sufficient for exception handling, however if you place a finally block then it will always run after the execution of try block.
- In normal case when there is no exception in try block then the finally block is executed after try block.
- An exception in the finally block, behaves exactly like any other exception.
- The statements present in the finally block execute even if the try block contains control transfer statements like return, break or continue.

Flow control in try catch finally

Control flow occurs in each of the given case

- **Control flow in try-catch clause OR try-catch-finally clause**
  - Case 1: Exception occurs in try block and handled in catch block
  - Case 2: Exception occurs in try-block is not handled in catch block
  - Case 3: Exception doesn’t occur in try-block
- **try-finally clause**
  - Case 1: Exception occurs in try block
  - Case 2: Exception doesn’t occur in try-block

Nested try catch block in Java

- When a try catch block is present in another try block then it is called the nested try catch block.
- Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.
- If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception.

Syntax of Nested try Catch

```
try {   statement 1;   statement 2;
  //try-catch block inside another try block
  try {   statement 3; statement 4;
    //try-catch block inside nested try block
    try {  statement 5; statement 6;  }
    catch(Exception e2) {  //Exception Message } }
  catch(Exception e1) { //Exception Message} }
//Catch of Main(parent) try block
catch(Exception e3) { //Exception Message }
```

User defined exceptions

- **We can define our own set of conditions or rules and throw an exception explicitly using throw keyword.**
- **For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword.**
- **Syntax of throw keyword:**
  - throw new exception_class("error message");
- **Example:**
  - throw new ArithmeticException("dividing a number by 5 is not allowed in this program");

Example

```
public class ThrowExampleDemo
{ static void checkEligibilty(int grescore)  {
    if(grescore<315) { throw new ArithmeticException("Student is not eligible for Admission"); }
    else {System.out.println("Student Entry is Valid!!"); }   }
  public static void main(String args[]){
   System.out.println("Welcome to the Admission process!!");
   checkEligibilty(300);
   System.out.println("Have a nice day.."); } }
```

Example

```
class MyException extends Exception
{   public MyException(String s) {   super(s); } }
public class ThrowExampleDemo1
{  void checkEligibilty(int grescore) throws MyException{
if(grescore<315){ throw new MyException("Student is not eligible for Admission"); }   }
   public static void main(String args[])
   {  ThrowExampleDemo1 obj = new ThrowExampleDemo1();
      try {   obj.checkEligibilty(60); }
      catch (MyException me)
      {  System.out.println("Caught the exception");
         System.out.println(me.getMessage()); } } }
```

Multithreading

- Multithreading refers to two or more tasks executing concurrently within a single program.
- A thread is an independent(sequential) path of execution within a program.
- Many threads can run concurrently within a program.
- Every thread in Java is created and controlled by the java.lang.Thread class.
- A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.

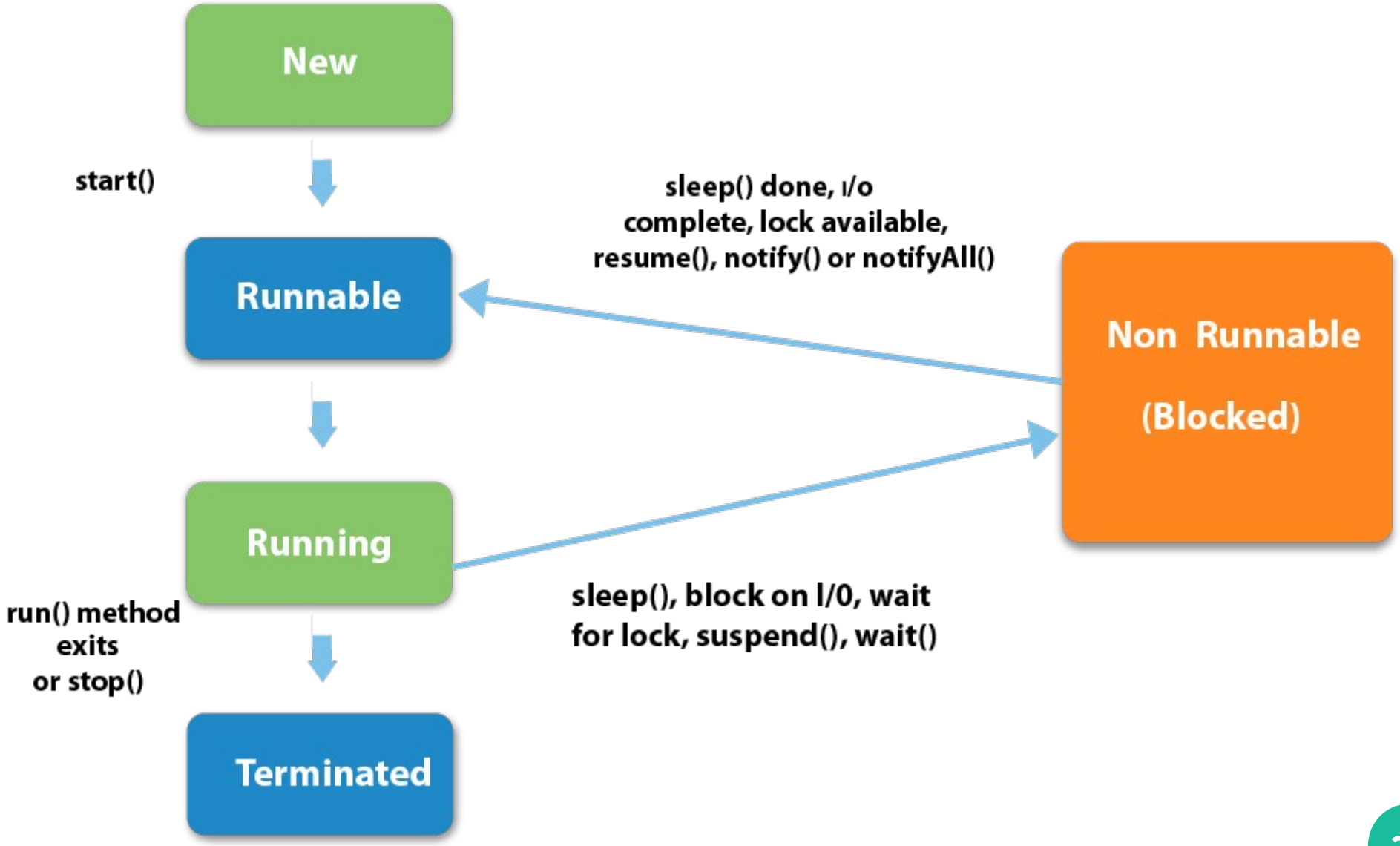
Advantages with Multithreading

- **Multithreading has several advantages over Multiprocessing**
  - Threads are lightweight compared to processes
  - Threads share the same address space and therefore can share both data and code
  - Context switching between threads is usually less expensive than between processes
  - Cost of thread intercommunication is relatively low that that of process intercommunication
  - Threads allow different tasks to be performed concurrently.

Thread Life Cycle (Thread States)

- The life cycle of the thread in java is controlled by JVM.
- The java thread states are as follows
  - New
  - Runnable
  - Running
  - Non-Runnable (Blocked)
  - Terminated

Thread Life Cycle (Thread States)



Thread Life Cycle (Thread States)

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.

Thread Creation

- **There are two ways to create thread in java**
  - Implement the Runnable interface (java.lang.Runnable)
  - By Extending the Thread class (java.lang.Thread)
- **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.

Thread Class

- **Commonly used Constructors of Thread class**
  - Thread()
  - Thread(String name)
  - Thread(Runnable r)
  - Thread(Runnable r,String name)
  - Thread(ThreadGroup group, Runnable r)
  - Thread(ThreadGroup group, Runnable r, String name)
  - Thread(ThreadGroup group, Runnable r, String name, long stackSize)
  - Thread(ThreadGroup group, String name)

Thread Class

- **Commonly used Methods of Thread class**
  - public void run(): is used to perform action for a thread.
  - public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
  - public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
  - public void join(): waits for a thread to die.
  - public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
  - public int getPriority(): returns the priority of the thread.
  - public int setPriority(int priority): changes the priority of the thread.
  - public String getName(): returns the name of the thread.

Thread Class

- **Commonly used Methods of Thread class.**
  - public void setName(String name): changes the name of the thread.
  - public Thread currentThread(): returns the reference of currently executing thread.
  - public int getId(): returns the id of the thread.
  - public Thread.State getState(): returns the state of the thread.
  - public boolean isAlive(): tests if the thread is alive.
  - public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
  - public void suspend(): is used to suspend the thread(depricated).

Thread Class

- **Commonly used Methods of Thread class.**
  - public void resume(): is used to resume the suspended thread(depricated).
  - public void stop(): is used to stop the thread(depricated).
  - public boolean isDaemon(): tests if the thread is a daemon thread.
  - public void setDaemon(boolean b): marks the thread as daemon or user thread.
  - public void interrupt(): interrupts the thread.
  - public boolean isInterrupted(): tests if the thread has been interrupted.
  - public static boolean interrupted(): tests if the current thread has been interrupted.

Extending a Thread Class

- A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.
- This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.
- The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

Example

```
class MyThread extends Thread {
MyThread() {  }
MyThread(String threadName)
{ super(threadName); // Initialize thread.
System.out.println(this);
start(); }
public void run()
{ //Display info about this particular thread
System.out.println(Thread.currentThread().getName());  }  }
```

Example

```
public class ThreadExample {
public static void main(String[] args) {
Thread thread1 = new Thread(new MyThread(), "thread1");
Thread thread2 = new Thread(new MyThread(), "thread2");
//The below 2 threads are assigned default names
Thread thread3 = new MyThread(); Thread thread4 = new MyThread();
Thread thread5 = new MyThread("thread5");
//Start the threads
thread1.start(); thread2.start();
thread3.start(); thread4.start();
try
{   Thread.currentThread().sleep(1000); } catch (InterruptedException e) { }
//Display info about the main thread
Thread.currentThread().setPriority(6);
System.out.println(Thread.currentThread()); } }
```

Thread[thread5,5,main]
thread5
thread1
Thread-2
thread2
Thread-3
Thread[main,6,main]

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.
- **Synatx**

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

Implementing a Runnable interface

- A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.
- An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
- The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned.
- The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception.

Example

- class RunnableThread implements Runnable {
- Thread runner;
- public RunnableThread() { }
- public RunnableThread(String threadName) {
- runner = new Thread(this, threadName); // (1) Create a new thread.
- System.out.println(runner.getName());
- runner.start(); // (2) Start the thread. }
- public void run() {//Display info about this particular thread
- System.out.println(Thread.currentThread().getName()); } }

Example

```
public class RunnableExample {
public static void main(String[] args) {
Thread thread1 = new Thread(new RunnableThread(), "thread1");
Thread thread2 = new Thread(new RunnableThread(), "thread2");
RunnableThread thread3 = new RunnableThread("thread3");
//Start the threads
thread1.start(); thread2.start();
try { Thread.currentThread().sleep(1000); }
catch (InterruptedException e) { }
//Display info about the main thread
Thread.currentThread().setPriority(3);
System.out.println(Thread.currentThread()); } }
```

```
thread3
thread3
thread1
thread2
Thread[main,3,main]
```

Thread Scheduler

- **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.**
- **The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.**
  - 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.

Multithreading in Java

- Multithreading in java is a process of executing multiple threads simultaneously.
- 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.

Multithreading in Java

- **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.

Example with Sleep Method

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

Example without Start Method

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

Thread Synchronization

- There are two types of thread synchronization mutual exclusive and inter-thread communication.

  1. Mutual Exclusive
     - Synchronized method.
     - Synchronized block.
     - static synchronization.
  2. 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.

Usage of Synchronized

```
class Table{
void printTable(int n){
  for(int i=1;i<=5;i++){
    System.out.println(n+" X "+i+" = "+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 ThreadSynchronizationDemo{
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 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.

Synchronized block in java

- **Syntax to use synchronized block**

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

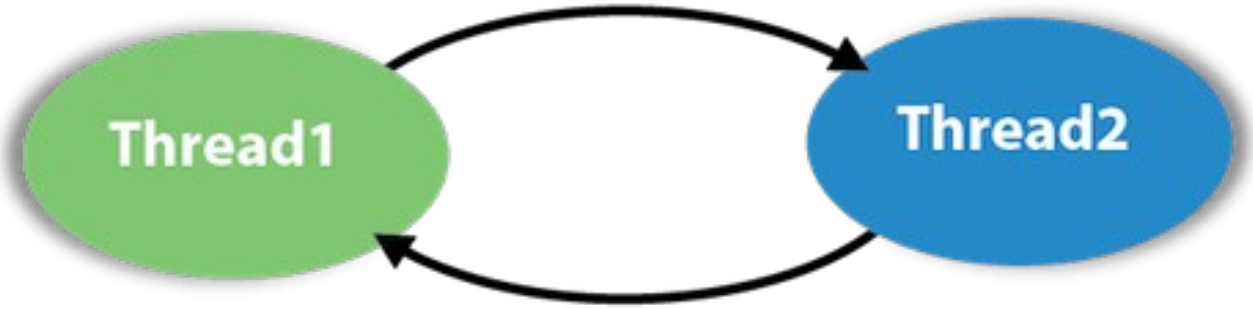
  - In the above example by adding the following one we can make it synchronized.
    - synchronized(this)

Static synchronization

- If you make any static method as synchronized, the lock will be on the class not on object.
- 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.
- If you want no interference between t1 and t3 or t2 and t4 then Static synchronization solves this problem.

Deadlock in java

- 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.



Example

```
public class ThreadDeadlockDemo {
 public static void main(String[] args) {
  final String resource1 = "Machine Learning";    final String resource2 = "Deep Learning";
  // 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");  } } }  };
  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();   }}
```

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()

Inter-thread communication in Java

- The Object class in Java has three final methods that allow threads to communicate about the locked status of a resource.

1. Wait

- It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
- The wait() method is actually tightly integrated with the synchronization lock,

Inter-thread communication in Java

- General Syntax of wait method

```
synchronized( lockObject )
{
   while( ! condition )
   {
      lockObject.wait();
   }
}
```

Inter-thread communication in Java

2. notify()

- It wakes up one single thread that called wait() on the same object.
- It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that can wake up.
- However, the lock is not actually given up until the notifier's synchronized block has completed.
- So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

Inter-thread communication in Java

- General syntax for calling notify() method is like this:

```
synchronized(lockObject)
{
   //establish_the_condition;
   lockObject.notify();
   //any additional code if needed
}
```

Inter-thread communication in Java

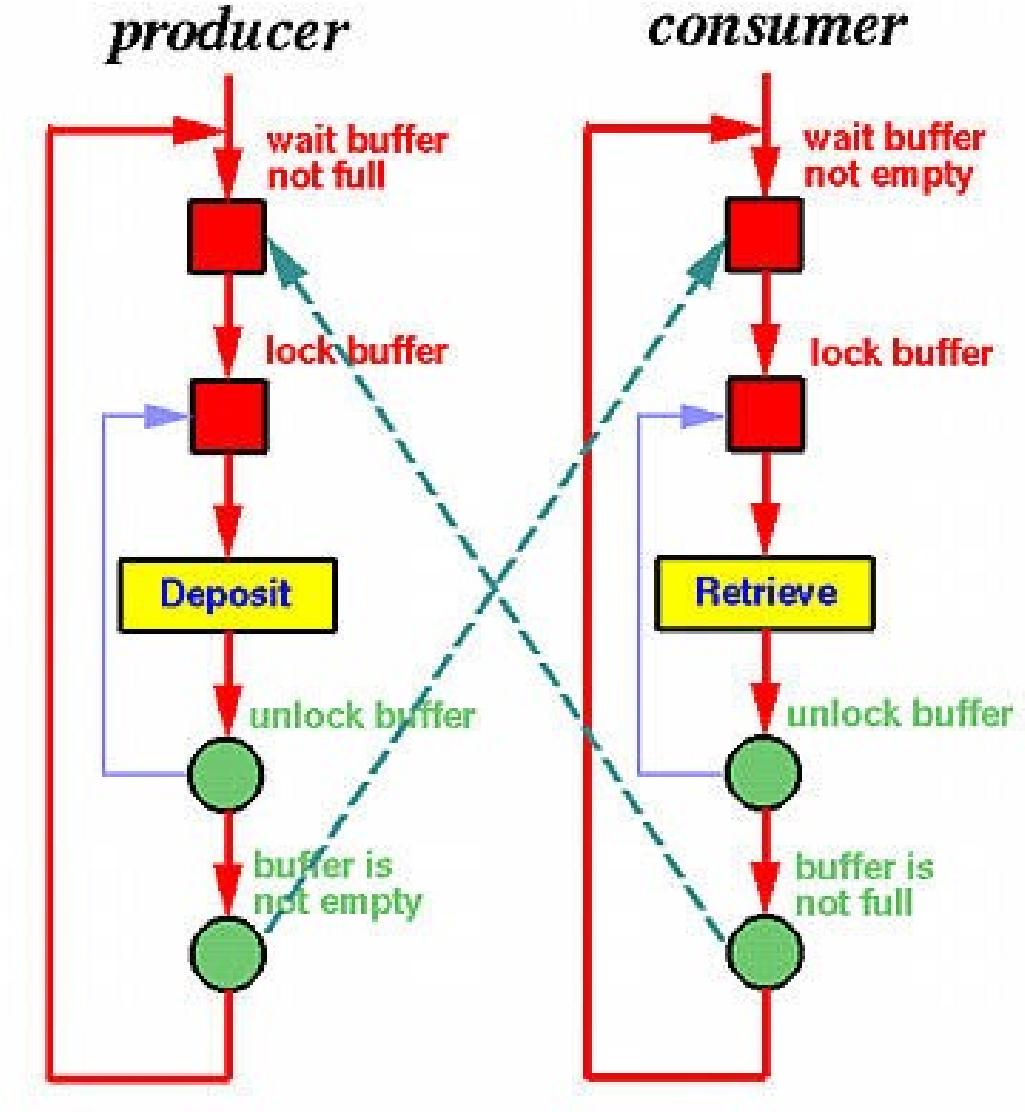
- **3. notifyAll()**
- It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify() method .
- General syntax for calling notify() method is like this:

```
synchronized(lockObject)
{
   //establish_the_condition;
   lockObject.notifyAll();
}
```

Inter-thread communication in Java

- **Producer-Consumer Problem**
  - Producer thread produce a new resource in every 1 second and put it in ‘taskQueue’.
  - Consumer thread takes 1 seconds to process consumed resource from ‘taskQueue’.
  - Max capacity of taskQueue is 5 i.e. maximum 5 resources can exist inside ‘taskQueue’ at any given time.
  - Both threads run infinitely.

Inter-thread communication in Java



Inter-thread communication in Java

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
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

References

- https://docs.oracle.com/javase/tutorial/
- https://www.javatpoint.com/features-of-java
- http://www.java2novice.com/java-fundamentals/
- http://archive.oreilly.com/oreillyschool/courses/java2/
- https://docs.oracle.com/javase/7/docs/api/
- https://www.javatpoint.com/
- https://beginnersbook.com/java-tutorial-for-beginners-with-examples/
- https://www.javaworld.com/
- https://www.geeksforgeeks.org/

Thank You!