Nadar Saraswathi College of Engineering & Technology

NSCET E-LEARNING PRESENTATION

LISTEN … LEARN… LEAD…

Electrical and Electronics Engineering

III YEAR/5th Semester

CS8392-OBJECT ORIENTED PROGRAMMING

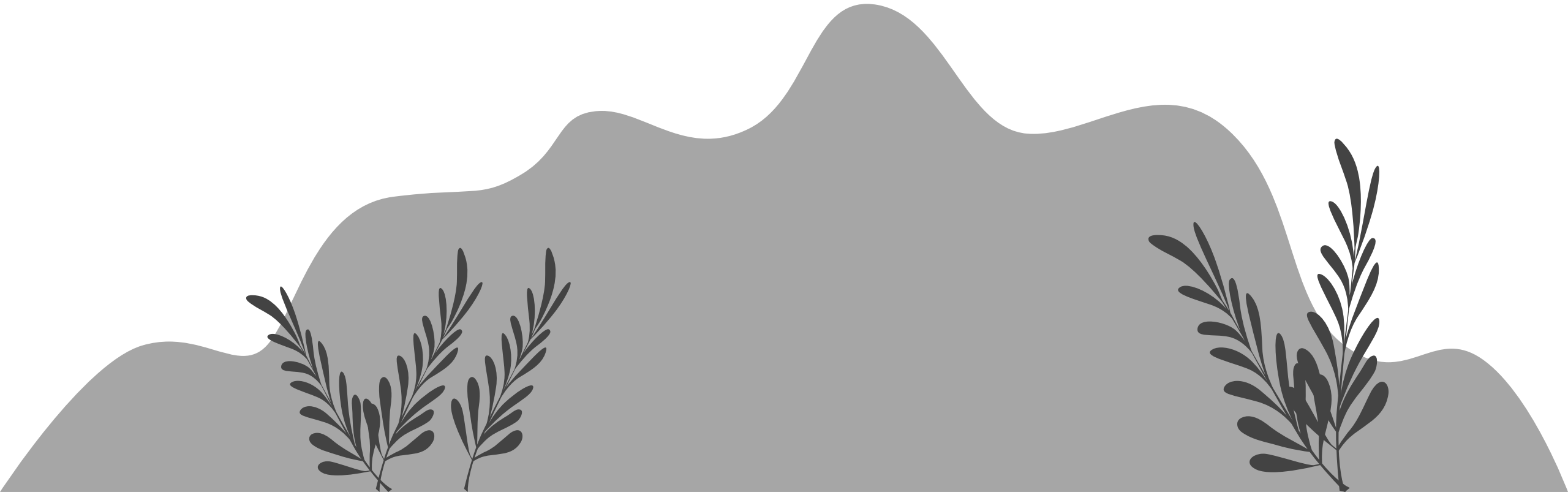
M.Gayathri B.Tech.,M.E.,

Assistant Professor

Nadar Saraswathi College of Engineering & Technology,

Vadapudupatti, Annanji (po), Theni – 625531.

UNIT IV -MULTITHREADING AND GENERIC PROGRAMMING



“**Learning** is not attained by chance, it must be sought for with ardor and attended to with diligence.”

—Abigail Adams

Unit IV - Syllabus

MULTITHREADING AND GENERIC PROGRAMMING

Differences between multi-threading and multitasking, thread life cycle, creating threads, synchronizing threads, Inter-thread communication, daemon threads, thread groups. Generic Programming – Generic classes – generic methods – Bounded Types – Restrictions and Limitations.

Multithreading and Multi-Tasking

- In programming, there are two main ways to improve the throughput of a program:

  i) by using multi-threading

  ii) by using multitasking

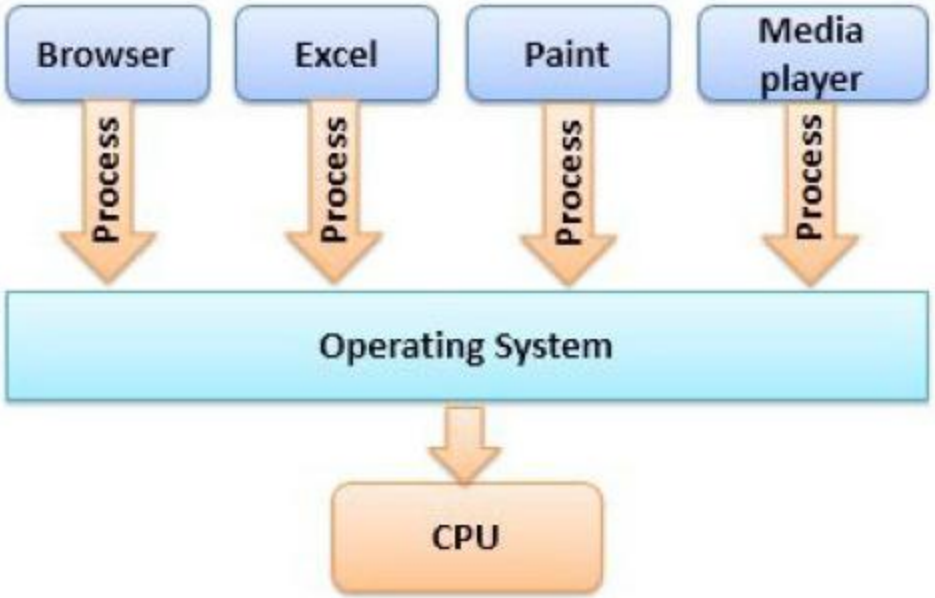
- Both these methods take advantage of parallelism to efficiently utilize the power of CPU and improve the throughput of program.

Differences between multi-threading and multitasking

Parameter	Multi Tasking	Multi Threading
Basic	Multitasking lets CPU to execute multiple tasks at the same time.	Multithreading lets CPU to execute multiple threads of a process simultaneously.
Switching	In multitasking, CPU switches between programs frequently.	In multithreading, CPU switches between the threads frequently.
Memory and Resource	In multitasking, system has to allocate separate memory and resources to each program that CPU is executing.	In multithreading, system has to allocate memory to a process, multiple threads of that process shares the same memory and resources allocated to the process.

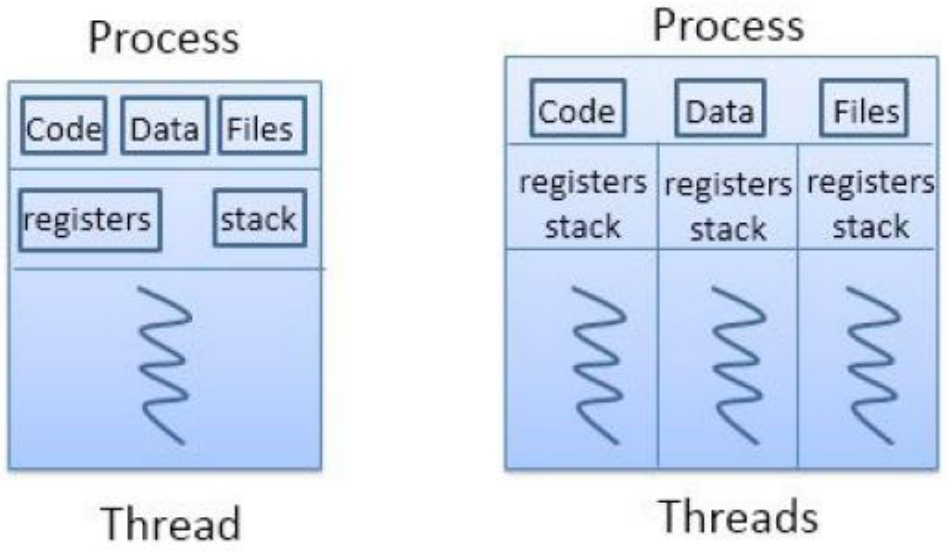
Multitasking

- Multitasking is when a single CPU performs several tasks (program, process, task, threads) at the same time. To perform multitasking, the CPU switches among these tasks very frequently so that user can interact with each program simultaneously.



Multi threading

- Multithreading is different from multitasking in a sense that multitasking allows multiple tasks at the same time, whereas, the Multithreading allows multiple threads of a single task (program, process) to be processed by CPU at the same time.



Benefits of Multithreading

- Multithreading increases the **responsiveness of system as, if one thread of the** application is not responding, the other would respond in that sense the user would not have to sit idle.
- Multithreading allows **resource sharing as threads belonging to the same process can** share code and data of the process and it allows a process to have multiple threads at the same time active in **same address space.**

Thread Lifecycle

- A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

  1) New

  2) Runnable

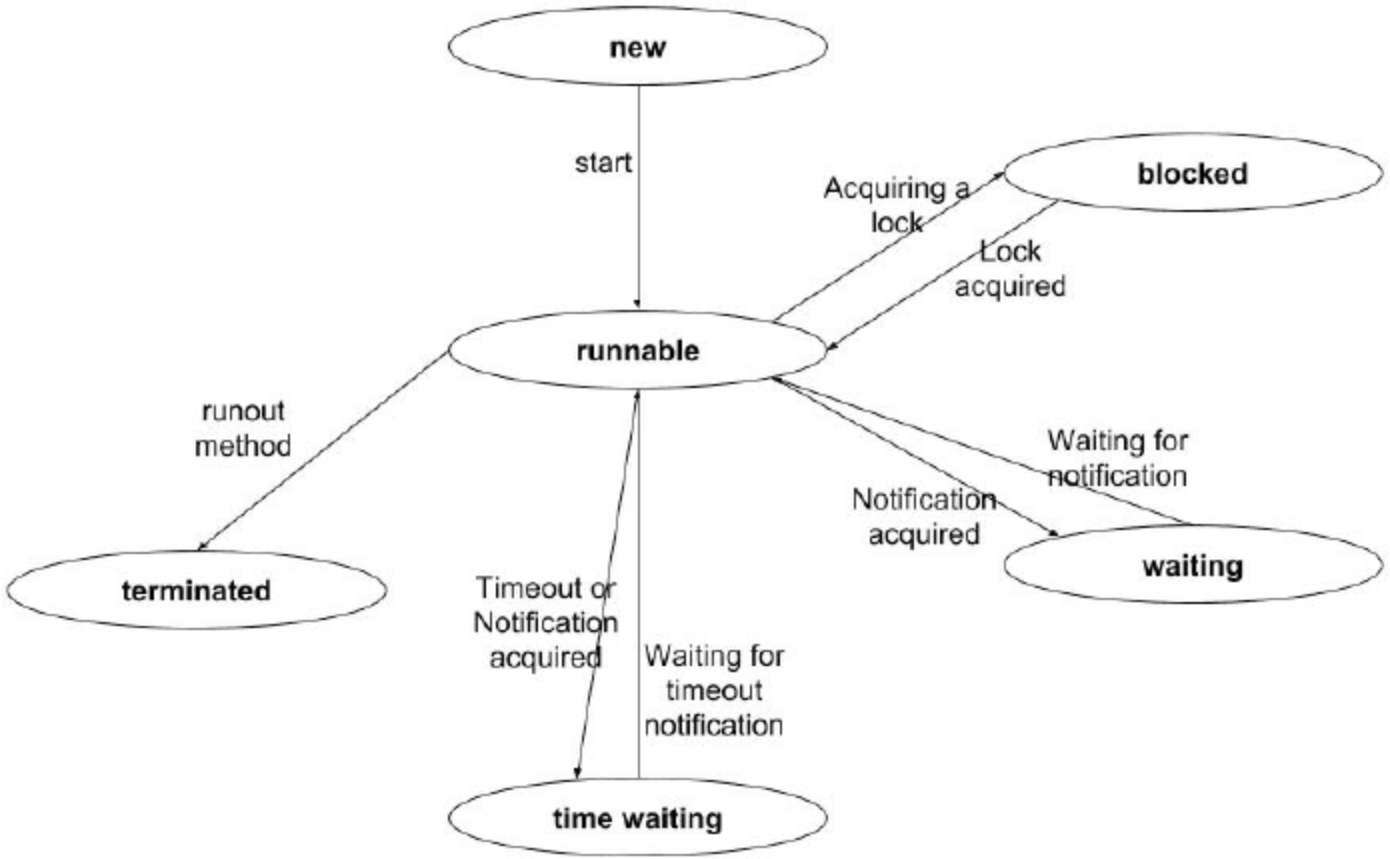
  3) Blocked

  4) Waiting

  5) Timed Waiting

  6) Terminated

The following figure represents various states of a thread at any instant of time:



Life Cycle of a thread

The various states of a thread at any instant of time:

1. New Thread:

2. Runnable State:

3. Blocked/Waiting state:

4. Timed Waiting:

5. Terminated State:

Creating Threads

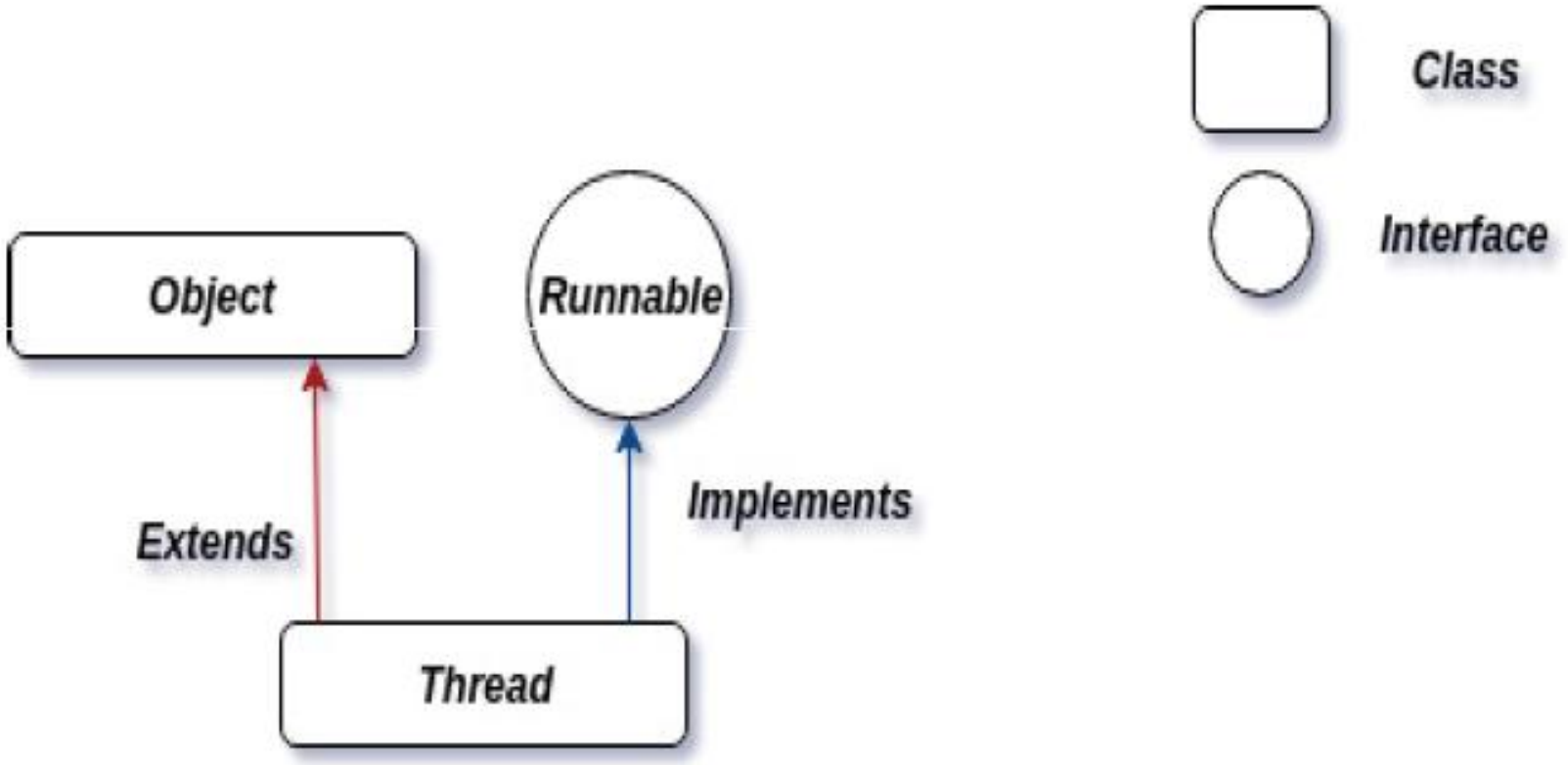
- Threading is a facility to allow multiple tasks to run concurrently within a single process. Threads are independent, concurrent execution through a program, and each thread has its own stack.

In Java, There are two ways to create a thread:

1) By extending Thread class.

2) By implementing Runnable interface.

Creating Threads



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

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r,String name)

Runnable interface

- Java program to create thread by implementing Runnable interface.

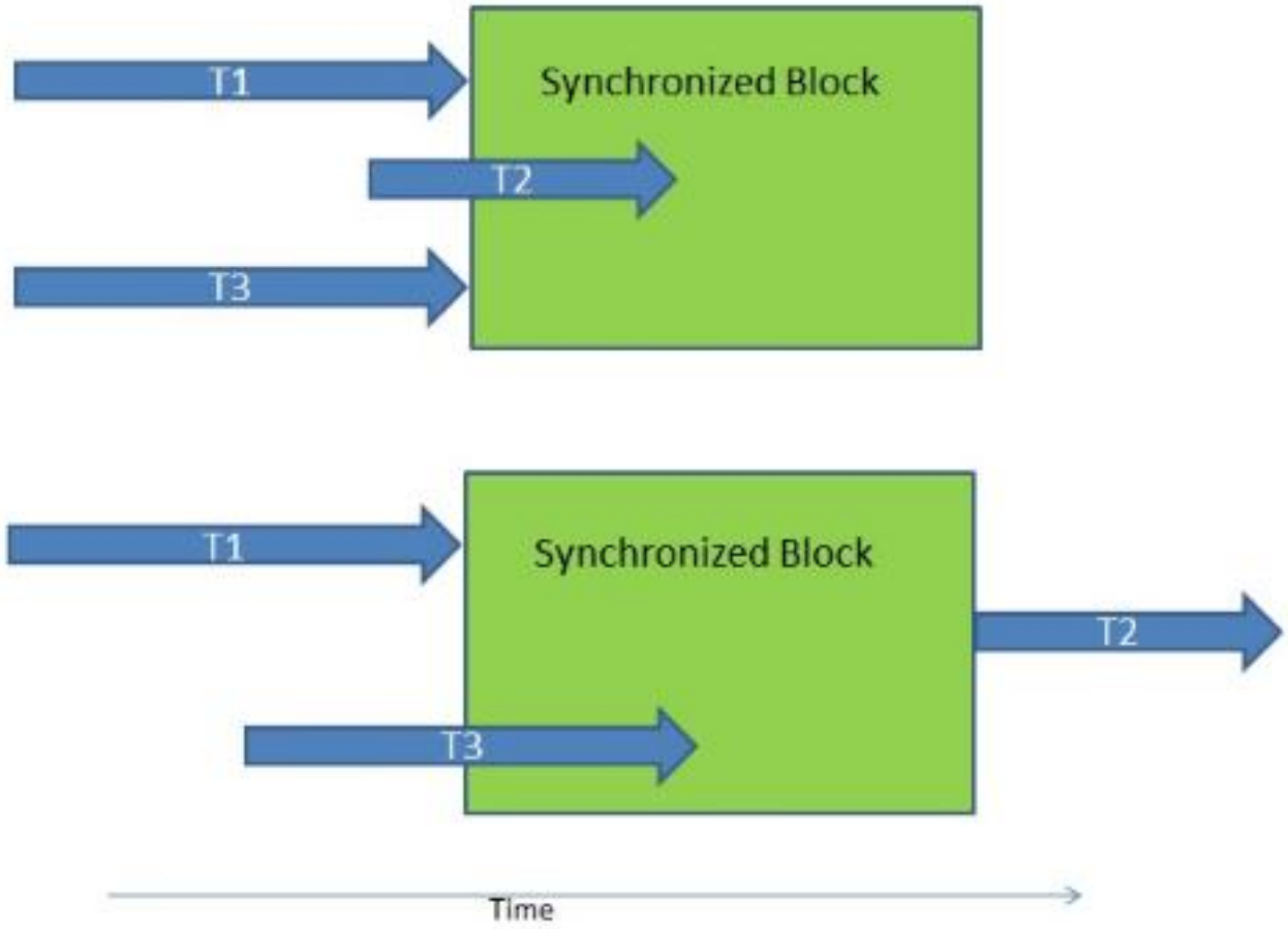
```
public class DemoRunnable implements
Runnable {
public void run() {
//Code
}
}
//start new thread with a "new Thread(new demoRunnable()).start()" call
```

Thread class

- Java program to create thread by extending Thread class

```
public class DemoThread extends Thread {

public DemoThread() {

super("DemoThread");

}

public void run() {

//Code

} }

//start new thread with a "new demoThread().start()"call
```

Thread Synchronization



Thread Synchronization

- 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

- Two types of synchronization

    1. Process Synchronization

    2. Thread Synchronization

Thread Synchronization

- **Two types of thread synchronization**
  - –mutual exclusive
  - – 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

Example -without Synchronization

```
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();
}  }
```

```
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.

Example -with Synchronization

```
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();
}  }
```

Output:

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

Inter-thread communication

- **Inter-thread communication** is allowing synchronized threads to communicate with each other.
- It 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.

Inter-thread communication

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.

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()

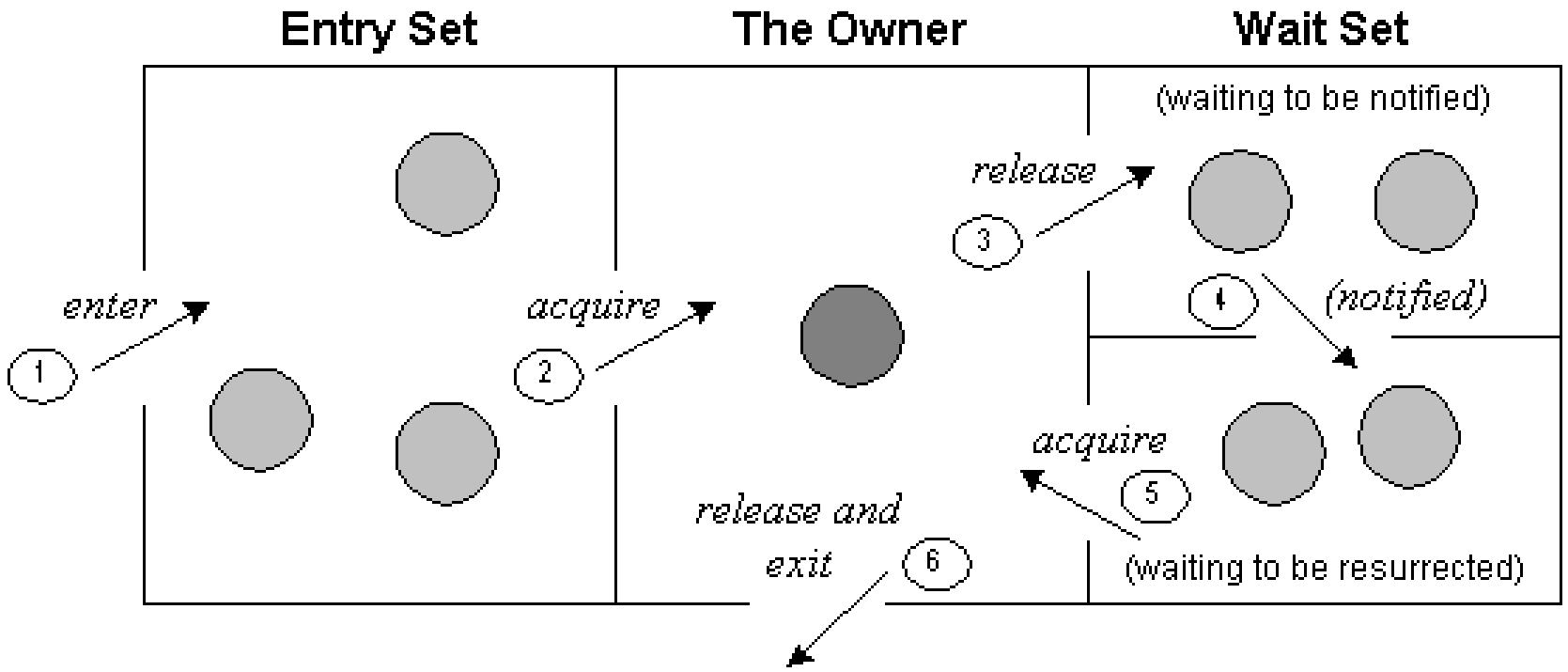
notifyAll() method

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

Syntax:

public final void notifyAll()

The process of inter-thread communication



Explanation of the diagram is as follows:

- 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.

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...");
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

Daemon Thread

- **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.
- You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Daemon Thread

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Methods for Java Daemon thread by Thread class

- The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

example of Daemon thread

```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }  }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();
  t1.setDaemon(true);//now t1 is daemon thread
  t1.start();//starting threads
  t2.start();
  t3.start();
 }  }
```

Output:

```
daemon thread
work user thread
work user thread work
```

ThreadGroup

- Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.
- Java thread group is implemented by *java.lang.ThreadGroup* class.
- A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

- There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

Code to group multiple threads

- ThreadGroup tg1 = **new** ThreadGroup("Group A");
- Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");
- Thread t2 = **new** Thread(tg1,**new** MyRunnable(),"two");
- Thread t3 = **new** Thread(tg1,**new** MyRunnable(),"three");

- Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.
- Now we can interrupt all threads by a single line of code only.
- Thread.currentThread().getThreadGroup().interrupt();

Example:

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
   public static void main(String[] args) {
     ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
        Thread t1 = new Thread(tg1, runnable,"one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable,"two");
        t2.start();
        Thread t3 = new Thread(tg1, runnable,"three");
        t3.start();
        System.out.println("Thread Group Name: "+tg1.getName());
       tg1.list();
          }          }
```

Output:

```
one
two
three
Thread Group Name: Parent
ThreadGroup
java.lang.ThreadGroup[name=Parent
ThreadGroup,maxpri=10]
Thread[one,5,Parent ThreadGroup]
Thread[two,5,Parent ThreadGroup]
Thread[three,5,Parent ThreadGroup]
```

Generic Programming

- Generic is a mechanism for creating a general model in which generic method and generic classes enable programmers to specify a single method(or a set o related methods) and single class (or a set of related classes) for performing and desired task.

Need for Generic

- Following features show the importance of generics in java:
  - ➢It saves the programmers burden of creating separate methods for handling data belonging to different data types.
  - ➢It allows the code reusability.
  - ➢Compact code can be created.

Generic class

- A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.
- Let's see a simple example to create and use the generic class.

Creating a generic class:

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

The T type indicates that it can refer to any type (like String, Integer, and Employee).The type you specify for the class will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
class TestGenerics3{

public static void main(String args[]){

MyGen<Integer> m=new MyGen<Integer>();

m.add(2);

//m.add("vivek");//Compile time error

System.out.println(m.get());

}}
```

Output:

2

Type Parameters

- The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:
- T - Type
- E - Element
- K - Key
- N - Number
- V - Value

Generic Method

- Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.
- Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{
    public static < E > void printArray(E[] elements) {
       for ( E element : elements){
          System.out.println(element );
        }
        System.out.println();
    }
    public static void main( String args[] ) {
       Integer[] intArray = { 10, 20, 30 };
       Character[] charArray = { 'J', 'A', 'V', 'A'};
       System.out.println( "Printing Integer Array" );
       printArray( intArray  );
       System.out.println( "Printing Character Array" );
       printArray( charArray );
    }  }
```

Output:

```
Printing Integer Array
10
20
30
Printing Character Array
J
A
V
A
```

Bounded Type

- While creating objects to generic classes we can pass any derived type as type parameters.
- Many times it will be useful to limit the types that can be passed to type parameters. For that purpose, bounded types are introduced in generics.
- Using bounded types, we can make the objects of generic.class to have data of specific derived types.

- For example, If we want a generic class that works only with numbers (like int, double, float, long ....) then declare type parameter of that class as a bounded type to Number class.
- Then while creating objects to that class you have to pass only Number types or is subclass types as type parameters.
- The **syntax** for declaring Bounded type parameters is
- <T extende SuperClass>
- This specifies that T' can only be replaced by 'SuperClass' or it's sub classes

For example

```
class Test<T extends Number> //Declaring Number class as upper bound of T
{
   Tt;
   public Test(T t) {
            this.t = t;
   }
   public T getT() {
            retum t;
   } }
public class BoundedTypeDemo {
public static void main(String|| args) {
//Creating object by passing Number as a type parameter
Test<Number> obj1 = new Test<Number>(123);
("The integer is: "+obj1.getT());
 //While Creating object by passing String as a type parameter, it gives compile time //error
   Test<String> obj2 = new Test<String>("I am string"): //Compile time error
   System.out.printin("The string is: "+obj2.getT();
}  }
```

Restrictions and Limitations

- 1. In Java, generic types are compile time entities. The runtime execution is possible only if it is used along with raw type.
- 2. Primitive type parameters is not allowed for generic programming. For example: Stack<int> is not allowed.
- 3. For the instances of generic class throw and catch instances are not allowed.

For example :

public class Test<T> extends Exception {

//code // Error:can't extend the Exception class

}

4. Instantiation of generic parameter T is not allowed.

For example :

new T() //Error

new T{10]

5. Arrays of parameterized types are not allowed.

For example :

new Stack<String>[10];//Error

6. Static fields and static methods with type parameters are not allowed.