Nadar Saraswathi College of Engineering & Technology

# NSCET

# E-LEARNING PRESENTATION

**LISTEN … LEARN… LEAD…**

# COMPUTER SCIENCE AND ENGINEERING

**II YEAR / III SEMESTER**

## CS8392 – OBJECT ORIENTED PROGRAMMING

**Mr.C.PRATHAP M.Tech.,(Phd).,**
**Assistant Professor**
**Nadar Saraswathi College of Engineering & Technology,**
**Vadapudupatti, Annanji (po), Theni – 625531.**

# UNIT 04
# MULTITHREADING AND GENERIC PROGRAMMING

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 area so saves memory, and context-switching between the threads takes less time than process.

MULTITHREADING

# 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, memory maps, updating lists, etc.

2) **Thread-based Multitasking (Multithreading)**

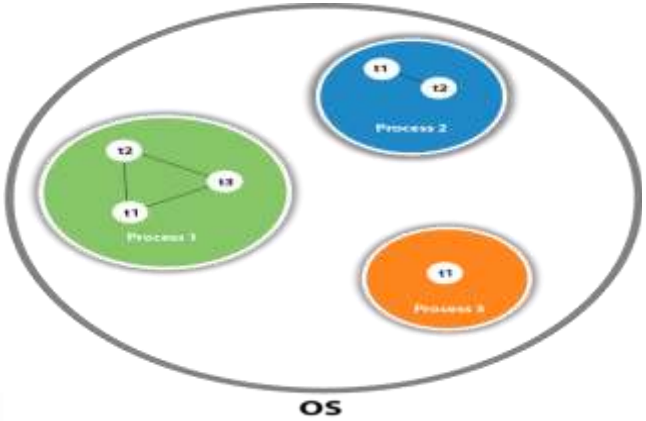
Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

# What is Thread in java

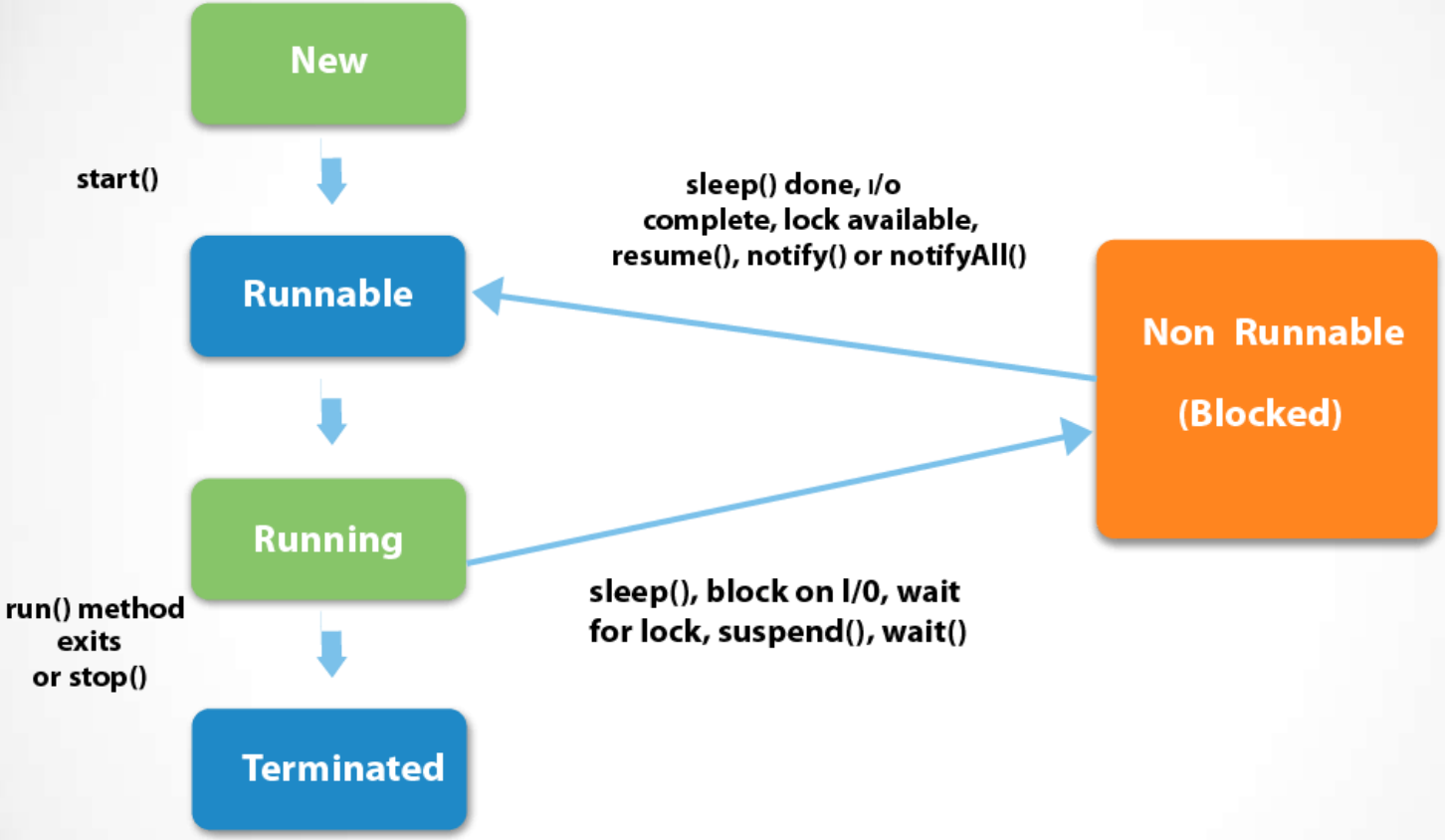
Thread is a lightweight subprocess, the smallest unit of processing.
It is a separate path of execution.
Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



# 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.

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated


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.

# 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 pre-emptive 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.o
ut.println(e);}
    System.out.println(i);  }}
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();
  t1.start();
  t2.start();
 } }
```

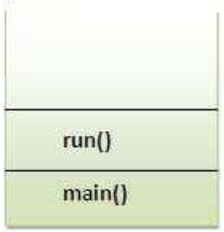
# Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

# What if we call run() method directly instead start() method?

Each thread starts in a separate call stack.

Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.




Stack
(main thread)

# The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

SYNTAX:

```
public void join()throws InterruptedException
public void join(long milliseconds)throws InterruptedException
```

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}
 t2.start();
 t3.start();
 }
}
```

```
class TestJoinMethod2 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i); }}
public static void main(String args[]){
 TestJoinMethod2 t1=new TestJoinMethod2();
 TestJoinMethod2 t2=new TestJoinMethod2();
 TestJoinMethod2 t3=new TestJoinMethod2();
 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}
 t2.start();
 t3.start(); }}
```

# getName(),setName(String) and getId() method:

```
lass TestJoinMethod3 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestJoinMethod3 t1=new TestJoinMethod3();
  TestJoinMethod3 t2=new TestJoinMethod3();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());
  System.out.println("id of t1:"+t1.getId());
  t1.start();
  t2.start();
  t1.setName("Sonoo Jaiswal");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

# The currentThread() method:

```
class TestJoinMethod4 extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 }
 public static void main(String args[]){
  TestJoinMethod4 t1=new TestJoinMethod4();
  TestJoinMethod4 t2=new TestJoinMethod4();
  t1.start();
  t2.start();
 }
}
```

# Naming Thread and Current Thread

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

**public String getName():** is used to return the name of a thread.

**public void setName(String name):** is used to change the name of a thread.

```
class TestMultiNaming1 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());
  t1.start();
  t2.start();
  t1.setName("Sonoo Jaiswal");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

# 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

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
  }
 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();
 }
}
```

# Daemon Thread in Java

**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.

```
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();
 }
}
```

# ThreadGroup in Java

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.

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.

```
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();
   }
  }
```

# Synchronization in Java

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

To prevent thread interference.

To prevent consistency problem.

# 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

```
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 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**

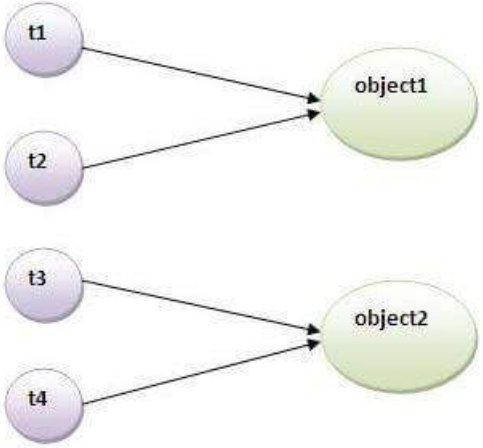
Synchronized block is used to lock an object for any shared resource.

Scope of synchronized block is smaller than the method.

```
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(); }}
```

# Static Synchronization

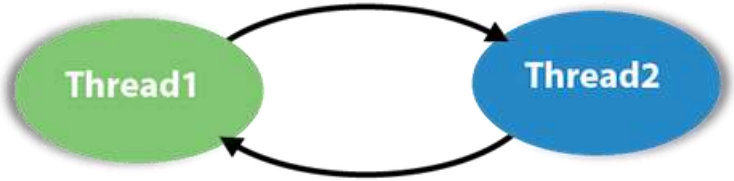
If you make any static method as synchronized, the lock will be on the class not on object.



```
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 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();
  }
}
```

# 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()

```
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();
}}
```

# Generic Programming

Generic programming enables the programmer to create classes,interfaces and methods that automatically works with all types of data(Integer, String, Float etc). It has expanded the ability to reuse the code safely and easily.

# Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:
1)Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
2)Type casting is not required: There is no need to typecast the object.
3)Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

# Generic class

A class that can refer to any type is known as generic class.
Generic class declaration defines set of parameterized type one for each possible invocation of the type parameters

# Generic Method

Like generic class, we can create generic method that can accept any type of argument.

# Bounded type

The type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter

**Syntax** :

<T extends superclass>

# Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

Cannot Instantiate Generic Types with Primitive Types

Cannot Create Instances of Type Parameters

Cannot Declare Static Fields Whose Types are Type Parameters

Cannot Use Casts or instanceof With Parameterized Types

Cannot Create Arrays of Parameterized Types

Cannot Create, Catch, or Throw Objects of Parameterized Types

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type