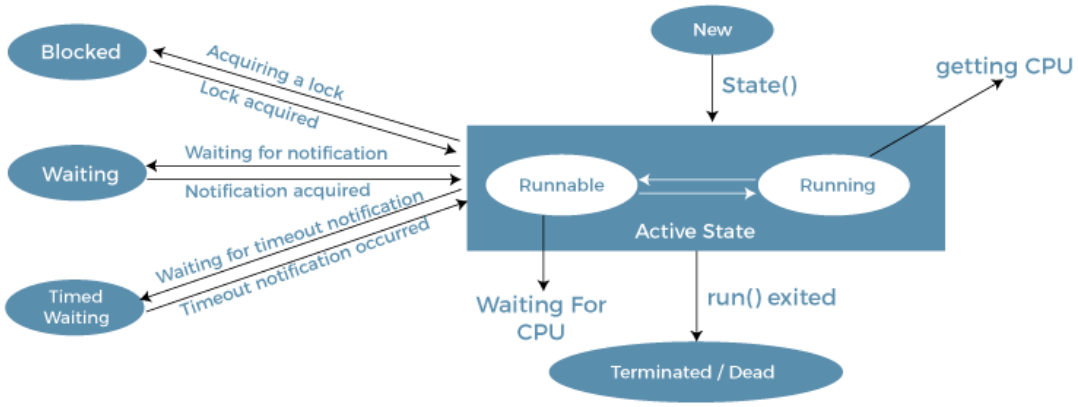
1. Threading

1.1. Introduction

- Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.
- Two distinct types of multitasking:
  - process-based:
    - process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
    - process-based multitasking deals with the "big picture,"
    - Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.
    - process-based multitasking is not under the control of Java.
  - thread-based:
    - thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
    - thread-based multitasking handles the details.
    - Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
    - Multithread multitasking is under the control of Java.
- Life cycle of a thread (thread states)



Life Cycle of a Thread

- In Java, a thread always exists in any one of the following states. These states are:
- New
  - Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.
- Active
  - When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.
  - Runnable: A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
  - Running: When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.
- Blocked / Waiting
  - Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.
- Timed Waiting
  - Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.
- Terminated
  - A thread reaches the termination state because of the following reasons:
    - When a thread has finished its job, then it exists or terminates normally.
    - Abnormal termination: It occurs when some unusual events such as an unhandled exception or segmentation fault.

1.2. Ways to define

- 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)
  - 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.
    - Etc.

- 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.

1.3. Instantiate and start a new thread

- Starting a thread:
  - The start() method of Thread class is used to start a newly created thread. It performs the 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.

Ex. 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();
 }
}
```

Output:

thread is running...

- Ex. 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);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output: thread is running...

Note: If you are not extending the Thread class, our class (Multi3) object would not be treated as a thread object. So you need to explicitly create the Thread class object. Here, we are passing the object of our class (Multi3) that implements Runnable so that our class run() method may execute.

1.4. Thread class constructor

1. Using the Thread Class: Thread() (Previous topic 1.3)
2. Using the Thread Class: Thread(Runnable r) (Previous topic 1.3)
3. Using the Thread Class: Thread(String Name)

Ex.

```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
```

```
}
}
```

Output: My first thread

4. Using the Thread Class: Thread(Runnable r, String Name)

Ex.

```
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}

// main method
public static void main(String argvs[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();

// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}
```

Output:

My new thread
Now the thread is running ...

1.5. Thread priority

- Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling).
- Setter & Getter Method of Thread Priority
  - public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.
  - public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method

throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

- 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.
- Ex.

```
import java.lang.*;
public class ThreadPriorityExample extends Thread
{
 public void run()
{
System.out.println("Inside the run() method");
}
 public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

 System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);

System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());
 // Main Thread
System.out.println("Currently Executing The Thread : " +
Thread.currentThread().getName());

System.out.println("Priority of the main thread is : " +
Thread.currentThread().getPriority());

Thread.currentThread().setPriority(10);
```

```
System.out.println("Priority of the main thread is : " +
Thread.currentThread().getPriority());
}
}
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

1.6. Yield()

- The yield() method of thread class causes the currently executing thread object to temporarily pause and allow other threads to execute.
- Ex.

```
public class JavaYieldExp extends Thread
{
  public void run()
  {
    for (int i=0; i<3 ; i++)
      System.out.println(Thread.currentThread().getName() + " in control");
  }
  public static void main(String[]args)
  {
    JavaYieldExp t1 = new JavaYieldExp();
    JavaYieldExp t2 = new JavaYieldExp();
    // this will call run() method
    t1.start();
    t2.start();
    for (int i=0; i<3; i++)
    {
      // Control passes to child thread
      t1.yield();
      System.out.println(Thread.currentThread().getName() + " in control");
    }
  }
}
```

Output:

```
main in control
main in control
```

```
main in control
Thread-0 in control
Thread-0 in control
Thread-0 in control
Thread-1 in control
Thread-1 in control
Thread-1 in control
```

1.7. Sleep()

- The method sleep() is being used to halt the working of a thread for a given amount of time.
- The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments.
- The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread.
- After the sleeping time is over, the thread starts its execution from where it has left.
- The sleep() Method Syntax:
    - public static void sleep(long mls) throws InterruptedException
    - public static void sleep(long mls, int n) throws InterruptedException
    - Parameters:
        - mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().
        - n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.
- Example of the sleep() method in Java : on the custom thread

Ex.

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

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

Output: 1 1 2 2 3 3 4 4

- Example of the sleep() Method in Java : on the main thread

Ex.

```
import java.lang.Thread;
import java.io.*;

public class TestSleepMethod2
{
   // main method
public static void main(String argvs[])
{
try {
for (int j = 0; j < 5; j++)
{

// The main thread sleeps for the 1000 milliseconds, which is 1 sec
// whenever the loop runs
Thread.sleep(1000);

// displaying the value of the variable
System.out.println(j);
}
}
catch (Exception expn)
{
// catching the exception
System.out.println(expn);
}
}
}
```

Output: 0 1 2 3 4

1.8. Join()

- When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state.
- The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state.
- Syntax:
    - public final void join() throws InterruptedException

- Example of join() Method in Java

Ex.

```
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.print(" "+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();
 }
}
```

Output:

1 2 3 4 5 1 1 2 2 3 3 4 4 5 5

We can see in the above example, when t1 completes its task then t2 and t3 starts executing.

1.9. Synchronization

- The process of allowing multiple threads to modify an object in a sequence is called synchronization. This is possible by using an object locking concept.
- Thread Synchronization is a process of allowing only one thread to use the object when multiple threads are trying to use the particular object at the same time. To achieve this Thread Synchronization we have to use a java keyword or modifier called "synchronized".
- General Syntax:

```
synchronized(objectidentifier)
{
      // Access shared variables and other shared resources;
}
```

Ex. Understanding the problem without synchronization

```
class Synchronization implements Runnable
{
   int tickets = 3;
   static int i = 1, j = 2, k = 3;
    void bookticket (String name, int wantedtickets)
   {
      if (wantedtickets <= tickets)
      {
```

```
            System.out.println (wantedtickets + " booked to " + name);
            tickets = tickets - wantedtickets;
          }
          else
          {
            System.out.println ("No tickets to book");
          }
        }
        public void run ()
        {
          String name = Thread.currentThread ().getName ();
          if (name.equals ("t1"))
          {
            bookticket (name, i);
          }
          else if (name.equals ("t2"))
          {
            bookticket (name, j);
          }
          else
          {
            bookticket (name, k);
          }
        }
        public static void main (String[]args)
        {
          Synchronization s = new Synchronization ();
          Thread t1 = new Thread (s);
          Thread t2 = new Thread (s);
          Thread t3 = new Thread (s);
          t1.setName ("t1");
          t2.setName ("t2");
          t3.setName ("t3");
          t1.start ();
          t2.start ();
          t3.start ();
        }
      }
```

Output:
1 booked to t1
2 booked to t2
3 booked to t3

Ex. Understanding the problem with synchronization

```
class Synchronization implements Runnable
{
   int tickets = 3;
   static int i = 1, j = 2, k = 3;
    synchronized void bookticket (String name, int wantedtickets)
   {
      if (wantedtickets <= tickets)
      {
         System.out.println (wantedtickets + " booked to " + name);
         tickets = tickets - wantedtickets;
      }
      else
      {
         System.out.println ("No tickets to book");
      }
   }
   public void run ()
   {
      String name = Thread.currentThread ().getName ();
      if (name.equals ("t1"))
      {
         bookticket (name, i);
      }
      else if (name.equals ("t2"))
      {
         bookticket (name, j);
      }
      else
      {
         bookticket (name, k);
      }
   }
   public static void main (String[]args)
   {
      Synchronization s = new Synchronization ();
      Thread t1 = new Thread (s);
      Thread t2 = new Thread (s);
      Thread t3 = new Thread (s);
      t1.setName ("t1");
      t2.setName ("t2");
      t3.setName ("t3");
      t1.start ();
      t2.start ();
      t3.start ();
```

```
    }
}
```

Outuput:
1 booked to t1
No tickets to book
2 booked to t2

1.10. Inter Thread communication

- Inter-thread communication is a process in which a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical region to be executed. i.e. synchronized threads communicate with each other.
- Below object class methods are used for Inter-thread communication process:
  - wait(): this method instructs the current thread to release the monitor held by it and to get suspended until some other threads sends a notification from the same monitor.
  - notify(): this method is used to send the notification to the thread that is suspended by the wait() method.
  - notifyAll(): this method is used to send the notification to all the threads that are suspended by wait() method.

Ex.

```
class Buffer{
    int a;
    boolean produced = false;

    public synchronized void produce(int x){
            if(produced){
                    System.out.println("Producer is waiting...");
                    try{
                            wait();
                    }catch(Exception e){
                            System.out.println(e);
                    }
            }
            a=x;
            System.out.println("Product" + a + " is produced.");
            produced = true;
            notify();
    }

    public synchronized void consume(){
            if(!produced){
                    System.out.println("Consumer is waiting...");
                    try{
                            wait();
```

```
                }catch(Exception e){
                        System.out.println(e);
                }
            }
            System.out.println("Product" + a + " is consumed.");
            produced = false;
            notify();
        }
    }

    class Producer extends Thread{
        Buffer b;
        public Producer(Buffer b){
            this.b = b;
        }

        public void run(){
            System.out.println("Producer start producing...");
            for(int i = 1; i <= 5; i++){
                b.produce(i);
            }
        }
    }

    class Consumer extends Thread{
        Buffer b;
        public Consumer(Buffer b){
            this.b = b;
        }

        public void run(){
            System.out.println("Consumer start consuming...");
            for(int i = 1; i <= 5; i++){
                b.consume();
            }
        }
    }

    public class ProducerConsumerExample {
        public static void main(String args[]){
            //Create Buffer object.
            Buffer b = new Buffer();

            //creating producer thread.
            Producer p = new Producer(b);

            //creating consumer thread.
            Consumer c = new Consumer(b);
```

```
            //starting threads.
            p.start();
            c.start();
        }
}
```

Output:

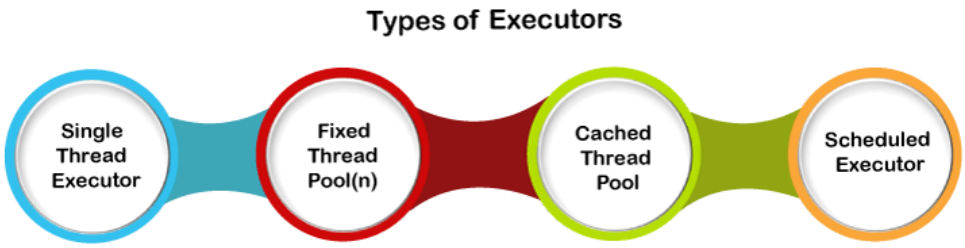
```
Consumer start consuming...
Consumer is waiting...
Producer start producing...
Product1 is produced.
Producer is waiting...
Product1 is consumed.
Consumer is waiting...
Product2 is produced.
Producer is waiting...
Product2 is consumed.
Consumer is waiting...
Product3 is produced.
Producer is waiting...
Product3 is consumed.
Consumer is waiting...
Product4 is produced.
Producer is waiting...
Product4 is consumed.
Consumer is waiting...
Product5 is produced.
Product5 is consumed.
```

1.11. Introduction of executor framework

- With the increase in the number of cores available in the processors nowadays, coupled with the ever-increasing need to achieve more throughput, multi-threading APIs are getting quite popular. Java provides its own multi-threading framework called the Java Executor Framework.
- Java executor framework (java.util.concurrent.Executor), released with the JDK 5 is used to run the Runnable objects without creating new threads every time and mostly re-using the already created threads.
- Now, the question which comes to our mind is why we have to create such thread pools when we already have the java.lang.Thread class for creating an object and Runnable/Callable interface for achieving parallelism by implementing them. So, the reason for creating such thread pools are as follows:
    - We need to create a large number of threads for adding a new thread without any throttling for each and every process. Due to which it requires more memory and cause wastage of resource. When each thread is swapped, the CPU starts to spend too much time.

- When we create a new thread for executing a new task cause overhead of thread creation. In order to manage this thread life-cycle, the execution time increase respectively.

- Types of Executors
  In Java, there are different types of executors available which are as follows:



- SingleThreadExecutor: The SingleThreadExecutor is a special type of executor that has only a single thread. It is used when we need to execute tasks in sequential order. In case when a thread dies due to some error or exception at the time of executing a task, a new thread is created, and all the subsequent tasks execute in that new one.

- FixedThreadPool(n): FixedThreadPool is another special type of executor that is a thread pool having a fixed number of threads. By this executor, the submitted task is executed by the n thread.

- CachedThreadPool: The CachedThreadPool is a special type of thread pool that is used to execute short-lived parallel tasks. The cached thread pool doesn't have a fixed number of threads. When a new task comes at a time when all the threads are busy in executing some other tasks, a new thread creates by the pool and add to the executor. When a thread becomes free, it carries out the execution of the new tasks. Threads are terminated and removed from the cached when they remain idle for sixty seconds.

- ScheduledExecutor: The ScheduledExecutor is another special type of executor which we use to run a certain task at regular intervals. It is also used when we need to delay a certain task.

2. Generics

2.1. Generic Methods

- It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.
- Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- You can write a single generic method declaration that can be called with arguments of different types.

- Ex.
  public class GenericMethodTest {

```
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
    for(E element : inputArray) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }

  public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array

    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

Output:

```
Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

2.2. Bounded Type Parameters

- There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter.

Ex.

```
// This class only accepts type parameters as any class
// which extends class A or class A itself.
// Passing any other type will cause compiler time error


class Bound<T extends A>
{
   private T objRef;

   public Bound(T obj){
      this.objRef = obj;
   }
    public void doRunTest(){
    this.objRef.displayClass();
   }
}

class A
{
   public void displayClass()
   {
      System.out.println("Inside super class A");
   }
}

class B extends A
{
   public void displayClass()
   {
      System.out.println("Inside sub class B");
   }
}

class C extends A
{
   public void displayClass()
   {
      System.out.println("Inside sub class C");
   }
}
```

```
public class BoundedClass
{
   public static void main(String a[])
   {
      // Creating object of sub class C and
      // passing it to Bound as a type parameter.
      Bound<C> bec = new Bound<C>(new C());
      bec.doRunTest();
      // Creating object of sub class B and
      // passing it to Bound as a type parameter.
      Bound<B> beb = new Bound<B>(new B());
      beb.doRunTest();
      // similarly passing super class A
      Bound<A> bea = new Bound<A>(new A());
      bea.doRunTest();

// Bound<String> bes = new Bound<String>(new String());
//bea.doRunTest();



   }
}
```

- Multiple bounds
  - Ex.

```
class Bound<T extends A & B>
{

        private T objRef;

        public Bound(T obj){
                this.objRef = obj;
        }
```

```
        public void doRunTest(){
            this.objRef.displayClass();
        }
}

interface B
{
        public void displayClass();
}

class A implements B
{
        public void displayClass()
        {
            System.out.println("Inside super class A");
        }
}

public class BoundedClass
{
        public static void main(String a[])
        {
            //Creating object of sub class A and
            //passing it to Bound as a type parameter.
            Bound<A> bea = new Bound<A>(new A());
            bea.doRunTest();

        }
}
```

2.3. wild char character(?)

- In generic code, the question mark (?), called the wildcard, represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type.

Ex.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.LinkedList;


public class WildCardSimpleExample {
 public static void printCollection(Collection<?> c) {
    for (Object e : c) {
       System.out.println(e);
```

```
    }
}

public static void main(String[] args) {
 Collection<String> collection = new ArrayList<>();
 collection.add("ArrayList Collection");
 printCollection(collection);
 Collection<String> collection2 = new LinkedList<>();
 collection2.add("LinkedList Collection");
 printCollection(collection2);
 Collection<String> collection3 = new HashSet<>();
 collection3.add("HashSet Collection");
 printCollection(collection3);

 }
}
```

Output:
ArrayList Collection
LinkedList Collection
HashSet Collection

2.4. comparable and comparator

- Comparable and Comparator both are interfaces and can be used to sort collection elements.
- Comparator interface sort collection using two objects provided to it, whereas comparable interface compares" this" refers to the one objects provided to it.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.

5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

- Ex. Without implementing comparable or comparator

```
import java.util.*;
class Student {
   private int id;
   private String name;
   private int age;
    public Student(int id, String name, int age) {
      this.id = id;
      this.name = name;
      this.age = age;
   }
    public int getId() {
      return id;
   }
    public String getName() {
      return name;
   }
    public int getAge() {
      return age;
   }
       public String toString() {
      return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age +"]";
   }

}

public class WithoutComparable {

   public static void main(String args[])
   {
      //Creating Student list
      ArrayList<Student> stuList = new ArrayList<Student>();

      //Adding Student elements
      stuList.add(new Student(1, "Shrawani",20));
      stuList.add(new Student(4, "Het", 19));
      stuList.add(new Student(3, "Vikash", 21));
      stuList.add(new Student(2, "Dhruv", 18));

      //Print array elements
```

```
        System.out.println("List before sorting: ");
        for(Student student : stuList){
           System.out.println(student);
        }

        //Sorting list elements
        Collections.sort(stuList);

        //Print array elements
        System.out.println("List after sorting: ");
        for(Student student : stuList){
           System.out.println(student);
        }
     }
  }
```

Output: Erro-> The method sort(List<T>) in the type Collections is not applicable for the arguments (ArrayList<Student>)

- Comparable interface: Comparable interface can be used for sorting operation on custom objects collection, which is defined in java.lang package. It has only one method that is compareTo(Object o). It returns a negative int value, zero, or a positive int value if “this” i.e. current object is less than, equal to, or greater than the object passed as an parameter/argument.

Ex.

```
import java.util.*;

class Student implements Comparable<Student> {
   private int id;
   private String name;
   private int age;

   public Student(int id, String name, int age) {
      this.id = id;
      this.name = name;
      this.age = age;
   }

   public int getId() {
      return id;
   }

   public String getName() {
      return name;
```

```
    }

    public int getAge() {
        return age;
    }

    @Override
    //Override compareTo method to customizing sorting algorithm
    public int compareTo(Student stu) {
        return (this.id - stu.id);
    }

    @Override
    //Override toString method to return Student details
    public String toString() {
        return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age +"]";
    }

}

public class ComparableExample {

    public static void main(String args[])
    {
        //Creating Student list
        ArrayList<Student> stuList = new ArrayList<Student>();

        //Adding Student elements
        stuList.add(new Student(1, "Shravani", 19));
        stuList.add(new Student(4, "Rushi", 20));
        stuList.add(new Student(3, "Parthiv", 21));
        stuList.add(new Student(2, "Madhu", 18));

        //Print array elements
        System.out.println("List before sorting: ");
        for(Student student : stuList){
            System.out.println(student);
        }

        //Sorting list elements
        Collections.sort(stuList);

        //Print array elements
        System.out.println("List after sorting: ");
        for(Student student : stuList){
```

```
            System.out.println(student);
        }
    }
}
```

- Comparator interface: Like Comparable interface, Comparator interface also used to sort a collection of custom objects and it is defined in java.util package. Its compare(Object obj1,Object obj2) method have to be implemented. The compare(Object obj1,Object obj2) method have to implemented in such a way that it will return a negative int value, zero, or a positive int value if first object/argument is less than, equal to, or greater than the second object/argument.

  Ex.

```
import java.util.*;
import java.io.*;

class Student {
   private int id;
   private String name;
   private int age;

   public Student(int id, String name, int age) {
      this.id = id;
      this.name = name;
      this.age = age;
   }

   public int getId() {
      return id;
   }

   public String getName() {
      return name;
   }

   public int getAge() {
      return age;
   }

   //Compare based on id
   public static Comparator<Student> IdComparator = new Comparator<Student>() {

      @Override
      public int compare(Student stu1, Student stu2) {
```

```
            return (int) (stu1.getId() - stu2.getId());
        }
    };

    //Compare based on name
    public static Comparator<Student> NameComparator = new Comparator<Student>() {

        @Override
        public int compare(Student stu1, Student stu2) {
            return stu1.getName().compareTo(stu2.getName());
        }
    };

    @Override
    //Override toString method to return Student details
    public String toString() {
        return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age +"]";
    }

}

public class ComparatorExample {

    public static void main(String args[])
    {
        //Creating Student list
        ArrayList<Student> stuList = new ArrayList<Student>();

        //Adding Student elements
        stuList.add(new Student(1, "Himanshi", 32));
        stuList.add(new Student(4, "Amani", 33));
        stuList.add(new Student(3, "Swati", 33));
        stuList.add(new Student(2, "Prabhjot", 31));

        //Print array elements
        System.out.println("List before sorting: ");
        for(Student student : stuList){
            System.out.println(student);
        }

        //Sorting list elements by Student Id
        Collections.sort(stuList, Student.IdComparator);

        //Print array elements
        System.out.println("List after sorting by Student Id: ");
```

```
        for(Student student : stuList){
           System.out.println(student);
        }

        //Sorting list elements by Student Name
        Collections.sort(stuList, Student.NameComparator);

        //Print array elements
        System.out.println("List after sorting by Student Name: ");
        for(Student student : stuList){
           System.out.println(student);
        }
     }
}
```