UNIT – 4

Illustrate the concept of synchronization in thread.

Java Synchronization is used to make sure by some synchronization method that only one thread can access the resource at a given point in time.

Thread Synchronization is used to coordinate and ordering of the execution of the threads in a multi-threaded program. There are two types of thread synchronization are mentioned below:

- Mutual Exclusive
- Cooperation (Inter-thread communication in Java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. There are three types of Mutual Exclusive mentioned below:

a) **Synchronized method.**
b) **Synchronized block.**
c) **Static synchronization.**

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 of java synchronized method

```
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 we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

If we put all the codes of the method in the synchronized block, it will work same as the synchronized 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();
} }
```

Illustrate in detail about multithread programming with example.

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Java Thread class

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

```
public class DisplayMessage implements Runnable {
  private String message;
    public DisplayMessage(String message) {
    this.message = message;   }

  public void run() {
    while(true) {
      System.out.println(message);
    }   }}

public class GuessANumber extends Thread {
  private int number;
  public GuessANumber(int number) {
    this.number = number;
  }
  public void run() {
    int counter = 0;
    int guess = 0;
    do {
      guess = (int) (Math.random() * 100 + 1);
      System.out.println(this.getName() + " guesses " + guess);
      counter++;
    } while(guess != number);
    System.out.println("** Correct!" + this.getName() + "in" + counter + "guesses.**");
  }}
```

Following is the main program, which makes use of the above-defined classes –

```
public class ThreadClassDemo {
  public static void main(String [] args) {
    Runnable hello = new DisplayMessage("Hello");
    Thread thread1 = new Thread(hello);
    thread1.setDaemon(true);
    thread1.setName("hello");
    System.out.println("Starting hello thread...");
    thread1.start();
     Runnable bye = new DisplayMessage("Goodbye");
    Thread thread2 = new Thread(bye);
    thread2.setPriority(Thread.MIN_PRIORITY);
    thread2.setDaemon(true);
    System.out.println("Starting goodbye thread...");
    thread2.start();
    System.out.println("Starting thread3...");
    Thread thread3 = new GuessANumber(27);
    thread3.start();
    try {
      thread3.join();
    } catch (InterruptedException e) {
      System.out.println("Thread interrupted.");
    }
    System.out.println("Starting thread4...");
    Thread thread4 = new GuessANumber(75);
        thread4.start();
    System.out.println("main() is ending...");
  }}
```

```
class DisplayMessage implements Runnable {
  private String message;
    public DisplayMessage(String message) {
    this.message = message;
  }
  public void run() {
    while(true) {
      System.out.println(message);
    }  }}
class GuessANumber extends Thread {
  private int number;
  public GuessANumber(int number) {
    this.number = number;
  }
  public void run() {
    int counter = 0;
    int guess = 0;
    do {
      guess = (int) (Math.random() * 100 + 1);
      System.out.println(this.getName() + " guesses " + guess);
      counter++;
    } while(guess != number);
    System.out.println("** Correct!" + this.getName() + "in" + counter + "guesses.**");
  }}
```

Explain the concept of throwing and catching exception in java.

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the exception object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

Let's see the example of throw IOException.

throw new IOException("sorry device error");

Example 1: Throwing Unchecked Exception

```
public class TestThrow1 {
  //function to check if person is eligible to vote or not
  public static void validate(int age) {
    if(age<18) {
      //throw Arithmetic exception if not eligible to vote
      throw new ArithmeticException("Person is not eligible to vote");
    }
    else {
      System.out.println("Person is eligible to vote!!");
    }    }
  //main method
  public static void main(String args[]){
    //calling the function
    validate(13);
    System.out.println("rest of the code...");
}  }
```

Example 2: Throwing Checked Exception

```
public class TestThrow2 {
  //function to check if person is eligible to vote or not
  public static void method() throws FileNotFoundException {
     FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
    BufferedReader fileInput = new BufferedReader(file);
       throw new FileNotFoundException();
     }
  //main method
  public static void main(String args[]){
    try
    {
      method();
    }
    catch (FileNotFoundException e)
    {
      e.printStackTrace();
    }
    System.out.println("rest of the code...");
} }
```

Explain briefly about user defined exceptions and stack trace elements in exception handling mechanisms.

Java provides us the facility to create our own exceptions which are basically derived classes of Exception. Creating our own Exception is known as a custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user needs. In simple words, we can say that a User-Defined Exception or custom exception is creating your own exception class and throwing that exception using the 'throw' keyword.

```
class MyException extends Exception {
  public MyException(String s)
```

```
  {
    // Call constructor of parent Exception
    super(s);
  }}
// A Class that uses above MyException
public class Main {
  // Driver Program
  public static void main(String args[])
  {
    try {
      // Throw an object of user defined exception
      throw new MyException("Error in line one");
    }
    catch (MyException ex) {
      System.out.println("Caught");
       // Print the message from MyException object
      System.out.println(ex.getMessage());
    }  }}
```

In Java, the stack trace is an array of stack frames. It is also known as stack backtrace (or backtrace). The stack frames represent the movement of an application during the execution of the program. It traces the locations where exception raised.

```
public class StackTraceExample
{
public static void main(String args[])
{
demo();
}
static void demo()
{
```

```
demo1();
}
static void demo1()
{
demo2();
}
static void demo2()
{
demo3();
}
static void demo3()
{
Thread.dumpStack();
} }
```

Illustrate how to implement runnable interface for creating and starting threads?

```
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();
```

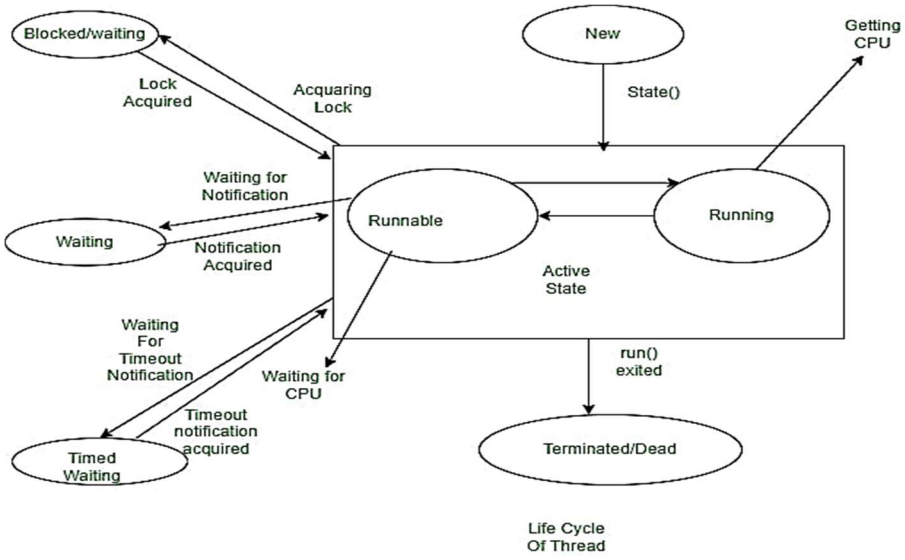
```
 // creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");


// 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);
}  }
```

Explain the states of a thread with a neat diagram.

There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

a) New State
b) Active State
c) Waiting/Blocked State
d) Timed Waiting State
e) Terminated State



We can see the working of different states in a Thread in the above Diagram, let us know in detail each and every state:

1. New State

By default, a Thread will be in a new state, in this state, code has not yet been run and the execution process is not yet initiated.

2. Active State

A Thread that is a new state by default gets transferred to Active state when it invokes the start() method, his Active state contains two sub-states namely:

Runnable State: In This State, The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.

Running State: When The Thread Receives CPU allocated by Thread Scheduler, it transfers from the "Runnable" state to the "Running" state. and after the expiry of its given time slice session, it again moves back to the "Runnable" state and waits for its next time slice.

3. Waiting/Blocked State

If a Thread is inactive but on a temporary time, then either it is a waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 needs to communicate to the camera and the other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in a Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

4. Timed Waiting State

Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing a Critical Coding operation and if it does not exist the CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

5. Terminated State

A thread will be in Terminated State, due to the below reasons:

Termination is achieved by a Thread when it finishes its task Normally.

Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions…etc. and such kind of Termination can be called Abnormal Termination.

A terminated Thread means it is dead and no longer available.

UNIT – 5

Explain how strings are handled in java? Explain with code, the creation of Substring, Concatenation and testing for equality.

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a'};

String s=new String(ch);
```

is same as:

```
String s="java";
```

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Creation of Substring

```
public class Substr1 {

  public static void main(String args[])

  {

    // Initializing String

    String Str = new String("Welcome to Java");

    System.out.print("The extracted substring is : ");

    System.out.println(Str.substring(3));

  }}
```

Concatenation of string

```
class TestStringConcatenation3{

 public static void main(String args[]){

  String s1="Sachin ";

  String s2="Tendulkar";

  String s3=s1.concat(s2);

  System.out.println(s3);//Sachin Tendulkar

 } }
```

Testing for equality of string

```
class Teststringcomparison4{
 public static void main(String args[]){
  String s1="Sachin";
  String s2="Sachin";
  String s3="Ratan";
  System.out.println(s1.compareTo(s2));//0
  System.out.println(s1.compareTo(s3));//1(because s1>s3)
  System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 } }
```

Illustrate the Java program to concatenate the two files and produce the output in the third file.

```
public class FileMerge
{
   public static void main(String[] args) throws IOException
   {
      // PrintWriter object for file3.txt
      PrintWriter pw = new PrintWriter("file3.txt");
            // BufferedReader object for file1.txt
      BufferedReader br = new BufferedReader(new FileReader("file1.txt"));
            String line = br.readLine();
            // loop to copy each line of
      // file1.txt to  file3.txt
      while (line != null)
      {
         pw.println(line);
         line = br.readLine();
      }
```

```
        br = new BufferedReader(new FileReader("file2.txt"));
              line = br.readLine();
               // loop to copy each line of
        // file2.txt to  file3.txt
        while(line != null)
        {
           pw.println(line);
           line = br.readLine();
        }
              pw.flush();
               // closing resources
        br.close();
        pw.close();
              System.out.println("Merged file1.txt and file2.txt into file3.txt");
    }
}
```

i)Explain I/O basics of Java(8)

Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.

System.in: This is the standard input stream that is used to read characters from the keyboard or any other standard input device.

System.out: This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.

Here is a list of the various print functions that we use to output statements:

print(): This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.

Syntax:

System.out.print(parameter);

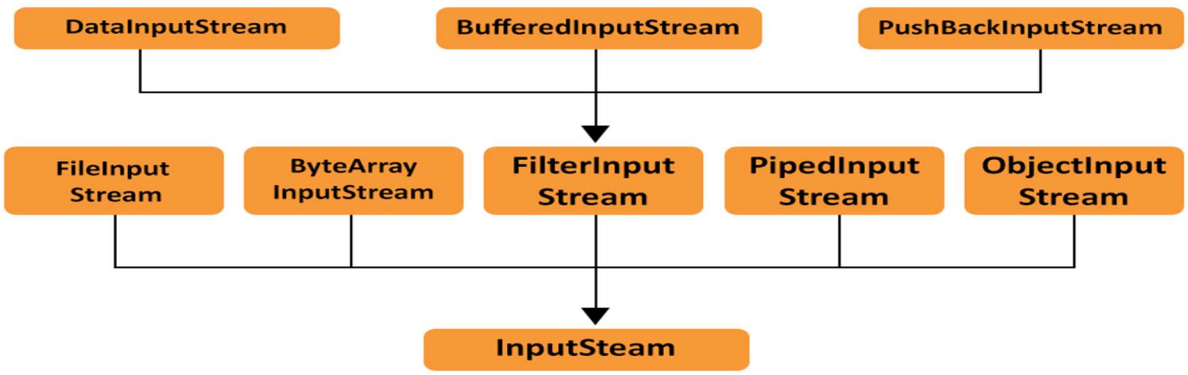
Example:

```
// Java code to illustrate print()
import java.io.*;
 class Demo_print {
   public static void main(String[] args)
   {
       // using print()
      // all are printed in the
      // same line
      System.out.print("GfG! ");
      System.out.print("GfG! ");
      System.out.print("GfG! ");
   }
}
```

ii)Explain predefine streams in Java(5)

In the programming world, a stream can be described as a series of data. It is known as a stream because it is similar to a stream of water that continues to flow. Java IO streams are flows of data that a user can either read from or write to. Like an array, a stream has no concept of indexing the read or write data. A stream is attached to a data origin or a data target.

1. InputStream – An input stream is used to read the data from a source in a Java application. Data can be anything, a file, an array, a peripheral device, or a socket. In Java, the class java.io.InputStream is the base class for all Java IO input streams.



Methods of Java IO InputStreams

read() – The read() method is used to read the next byte of data from the Input Stream. The value byte is passed on a scale of 0 to 255. If no byte is free because the end of the stream has arrived, the value -1 is passed.

mark(int arg) – The mark(int arg) method is used to mark the current position of the input stream. It sets read to limit, i.e., the maximum number of bytes that can be read before the mark position becomes invalid.

reset() – The reset() method is invoked by mark() method. It changes the position of the input stream back to the marked position.

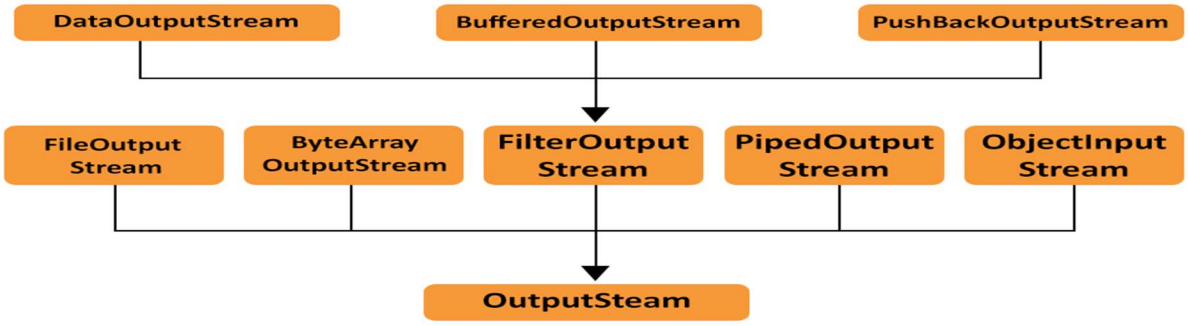
close() – The close() method is used to close the input stream and releases system resources associated with this stream to Garbage Collector.

read(byte [] arg) – The read(byte [] arg) method is used to read the number of bytes of arg.length from the input stream to the buffer array arg. The bytes read by read() method are returned as an int. If the length is zero, then no bytes are read, and 0 is returned unless there is an effort to read at least one byte.

skip(long arg) – The skip(long arg) method is used to skip and discard arg bytes in the input stream.

markSupported() – The markSupported() method tests if the inputStream supports the mark and reset methods. The markSupported method of Java IO InputStream yields false by default.

2. OutputStream – An output stream is used to write data (a file, an array, a peripheral device, or a socket) to a destination. In Java, the class java.io.OutputStream is the base class for all Java IO output streams.



Methods of Java IO OutputStreams

flush() – The flush() method is used for flushing the outputStream This method forces the buffered output bytes to be written out.

close() – The close() method is used to close the outputStream and to release the system resources affiliated with the stream.

write(int b) – The write(int b) method is used to write the specified byte to the outputStream.

write(byte [] b) – The write(byte [] b) method is used to write bytes of length b.length from the specified byte array to the outputStream.

i)Explain the file operations of READ from a file, WRITE to a file in Java with example program(8)

READ from a file,

```
// importing the FileReader class
import java.io.FileReader;
class Main {
  public static void main(String[] args) {
    char[] array = new char[100];
    try {
      // Creates a reader using the FileReader
      FileReader input = new FileReader("input.txt");
      // Reads characters
      input.read(array);
      System.out.println("Data in the file:");
      System.out.println(array);
      // Closes the reader
      input.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    } }}
```

WRITE to a file

```
// importing the FileWriter class
import java.io.FileWriter;
 class Main {
   public static void main(String args[]) {
     String data = "This is the data in the output file";
```

```
try {
  // Creates a Writer using FileWriter
  FileWriter output = new FileWriter("output.txt");
  // Writes string to the file
  output.write(data);
  System.out.println("Data is written to the file.");
  // Closes the writer
  output.close();
}
catch (Exception e) {
  e.getStackTrace();
} }}
```

ii)List the methods of File class in Java(5)

Method	**Description**
createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
canRead()	It tests whether the application can read the file denoted by this abstract pathname.
isAbsolute()	It tests whether this abstract pathname is absolute.
isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
getName()	It returns the name of the file or directory denoted by this abstract pathname.

getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
toURI()	It constructs a file: URI that represents this abstract pathname.
listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
mkdir()	It creates the directory named by this abstract pathname.

Explain with a program to show generic class and methods.

Generics means parameterized types. The idea is to allow a type (like Integer, String, etc., or user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as a class, interface, or method that operates on a parameterized type is a generic entity.

Types of Java Generics

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

```
class Test<T> {
   // An object of type T is declared
   T obj;
   Test(T obj) { this.obj = obj; } // constructor
   public T getObject() { return this.obj; }
}
// Driver class to test above
class Main {
```

```
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());
        // instance of String type
        Test<String> sObj
            = new Test<String>("Java");
        System.out.println(sObj.getObject());
    }
}
```

Generic Classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, ?are known as parameterized classes or parameterized types.

```
class Test {
    // A Generic method example
    static <T> void genericDisplay(T element)
    {
        System.out.println(element.getClass().getName()
                           + " = " + element);
    }
    // Driver method
    public static void main(String[] args)
    {
        // Calling generic method with Integer argument
        genericDisplay(11);
        // Calling generic method with String argument
        genericDisplay("Java");
```

```
    // Calling generic method with double argument
    genericDisplay(1.0);
  }}
```

i) Illustrate the motivations of generic programming. (7)

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

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.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error

String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);
 Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

ii.)List the rules to define a generic method.(6)

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

```
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

  }}
```