B.Sc (Computer Science)
Programming in Java
Unit-III

1. What is an Error? Explain the different types of Errors

It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing to program to produce unexpected results. Errors are the wrongs that can make a program go wrong. In computer terminology errors may be referred to as bugs. An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible errors.

Types of Errors: Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

Compile-Time Errors: All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the .class file. It is necessary to fix these errors to get it compiled. It becomes an easy job to a programmer to correct these errors because Java compiler tells us where the errors are located.
The most common errors are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables Incompatible types in assignments / initialization
- Bad references to objects
- Use of = in place of = = operator

Run-Time Errors: Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.
Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference to access a method or a variable.
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string

2. Explain Exception in Java.

Exception: An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e., informs us that an error has occurred).

If the object is not caught and handled properly, the interpreter will display an error message and will terminate (stop) the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate (proper) message for taking corrective actions. This task is known as **Exception Handling.**

The purpose of exception handling mechanism is to provide a means to detect (find) and report an *exceptional circumstance* so that appropriate can be taken. The mechanism suggests incorporation (to include) of a separate error handling code that performs the following tasks:

- Find the problem (*Hit* the exception)
- Inform that an error has occurred (*Throw* the exception)
- Receive the error information (*Catch* the exception)
- Take corrective actions (*Handle* the exception)

3. Explain Pre-defined Exceptions in Java.

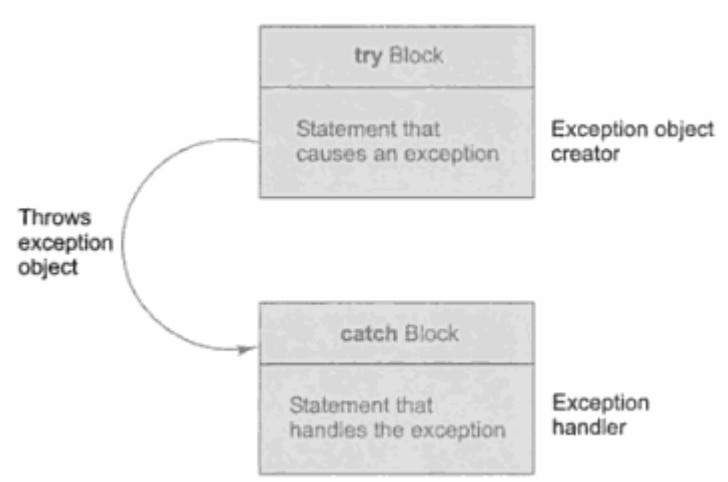
Some common exceptions that we must watch out for catching are listed below:

- **ArithmeticException** – Caused by math errors such as division by zero
- **ArrayIndexOutOfBoundsException** – Caused by bad array indexes
- **ArrayStoreException** – Caused when a program tries to store the wrong type of data in an array
- **FileNotFoundException** – Caused by an attempt to access a nonexistent file
- **IOException** – Caused by general I/O failures, such as inability to read from a file
- **NullPointerException** – Caused by referencing a null object
- **NumberFormatException** – Caused when a conversion between strings and number fails
- **OutOfMemoryException** – Caused when there's not enough memory to allocate a new object.
- **SecurityException** – Caused when an applet tries to perform an action not allowed by the browser's security setting
- **StackOverflowException** – Caused when the system runs out of stack space.

4. Explain the process of Exception Handling in Java.

Exception Handling: If the object is not caught and handled properly, the interpreter will display an error message and will terminate (stop) the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate (proper) message for taking corrective actions. This task is known as **Exception Handling.**

The basic concept of exception handling are throwing an exception and catching it.



Java uses a keyword *try* to preface a block of code that is likely to cause an error condition and *throw* an exception. A catch block defined by the keyword *catch* catches the exception *thrown* by the try block and handles it appropriately. The catch block is added immediately after the try block. The following syntax illustrates the use of simple *try* and *catch* statements:

```
…….
try
{
        statement;      // generates an exception
}
catch (Exception-type e)
{
        statement;      // process the exception
}
……..
……..
```

try & catch: The try block can have one or more statements that could generate an exception. If any statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch blocks that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every *try* statement should be followed by *at least one* catch statement; otherwise compilation error will occur.

The *catch* statement works like a method definition. The *catch* statement is passed a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

5. Write a program to handle division by zero exception.

```
class ZeroTest
```

```
{
        public static void main(String as[])
        {
                int a=5;
                int b=0;
                        try
                        {
                                System.out.println("Division="+ (a/b));
                        }
                        catch(ArithmeticException e)
                        {
                                System.out.println("Division by zero is not possible");
                        }
        }
}
```

6. How many catch blocks can we use with one try block?

It is possible to have more than one catch statement in the catch block. When an exception in a *try* block is generated, the Java treats the multiple *catch* statements like case in a *switch* statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

Example: *catch (Exception e);*

The *catch* statement simply ends with a semicolon, which does nothing. This statement will catch an exception and then ignore it.

Syntax:

```
……….
try
{
        statement;      // generates an exception
}
catch (Exception-Type-1 e)
{
        statement;      // process exception type 1
}
catch (Exception-Type-2 e)
{
        statement;      // process exception type 2
}
catch (Exception-Type-N e)
{
        statement;      // process exception type N
}
…….
```

7. Explain throwing our own exception in Java.

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows: **throw** new **Throwable_subclass;**

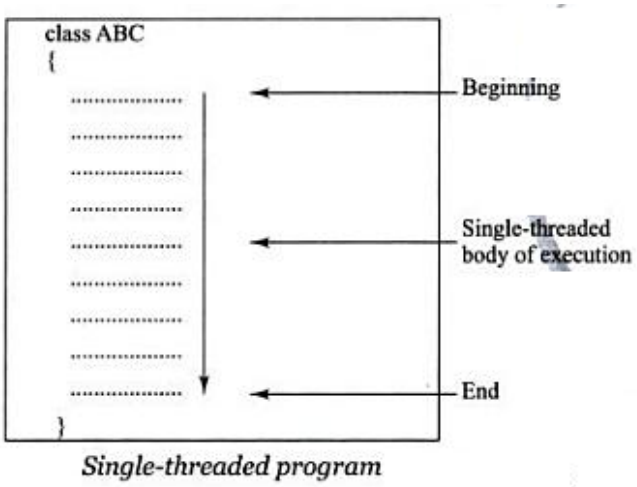
Example: throw new ArithmeticException( );
throw new NumberFormatException( );

```
import java.lang.Exception;
class MyException extends Exception
{
        MyException(String message)
        {
        super(message);
        }
}
class TestMyException
{
        public static void main(String args[ ])
        {
                int x = 5, y = 1000;
                try
                {
                float z = (float)x/(float)y;
                        if(z < 0.01)
                        {
                        throw new MyException("Number is too small");
                        }
                }
                catch(MyException e)
                {
                System.out.println("caught my exception");
                System.out.println(e.getMessage( ));
                }
                finally
                {
                System.out.println("I am always here");
                }
        }
}
```

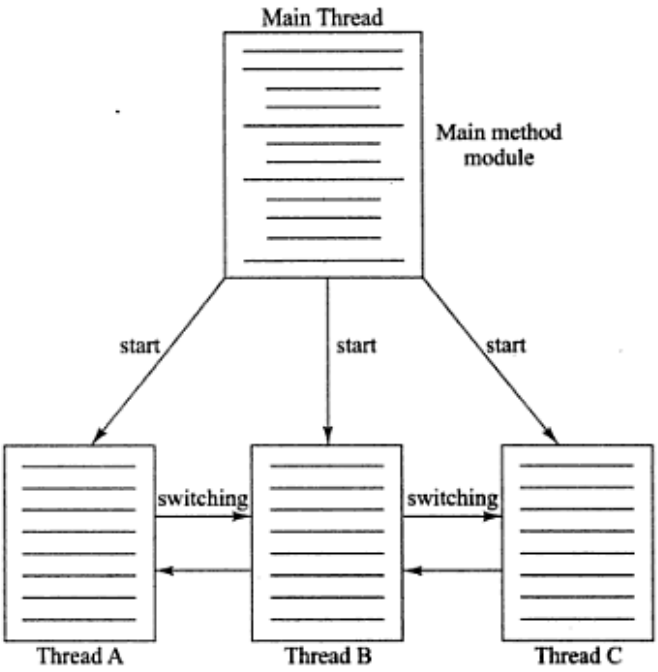
8. What is Thread?

Thread is a task or flow of execution that can be made to run using time-sharing principle. It is important to remember that 'threads running in parallel' does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads.

The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.



Single-threaded program



Multithreaded program

A **Thread** is similar to a program that has a single flow of control. It has a beginning, a body and an end, and executes commands sequentially. A normal java programming is a single-threaded program. That means, every java program will have atleast one thread.

9. What is the concept of Multi-Threading?

Modern operating systems such as Windows 7,8,10 & 11 can execute several programs simultaneously. This ability is known as **Multi-Tasking**. In other words, this is called as **Multi-Processing**.

In Java Terminology, this is called as **Multi-Threading**. It is a conceptual programming paradigm where a program (process) is divided into two or more sub-programs (processes), which can be implemented at the same time in parallel. It is a powerful programming tool that makes Java distinctly different from its other programming languages.

Light-weight Process: A thread is similar to a separate process, in that it can run independently of other threads. But it is lightweight, since the operating system doesn't have to give it its own memory space, and it shares memory with the other threads in the process. It runs within the context of a program because threads or sub-programs of a main application program.

Heavy-weight Process: In heavyweight processes in between threads that belong to different programs. They demand separate memory.

10. Explain the process of creation of Threads.

Creating threads in java is simple. Threads are implemented in the form of objects that contain a method called **run()**.The run() method is the heart of any thread. It is the only method in which the thread's behavior can be implemented.

```
public void run ( )
{
    ... // Thread code
}
```

The **run()** method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another method called **start( ).**

A new thread can be created by two ways:

- **By creating a Thread class**: Define a class that **extends Thread class** and override its run() method with the code required by the thread.
- **By converting a class to thread**: Defines a class that **implements Runnable interface**. The Runnable interface has only one method run(), that is to be defined in the method with the code to be executed by the thread.

11. How to extend Thread class?

Extending Thread Class: we create a thread class that extends *Thread* class and override its run() method with the code required by the thread.

To do this consider following steps:

- Declare the class that extends the Thread class.

```
public class MyThread extends Thread
{
}
```

- Implements the run() method that is responsible for executing the sequence of code that the thread will execute.

```
public void run()
{
  //Thread code
```

```
}
```

- create a thread object and call the start() method to initiate the thread execution

```
MyThread mt=new MyThread();
mt.start();
```

Example:

```
public class MyThread extends Thread
{
   public void run()
   {
      System.out.println("Hello from a thread!");
   }

   public static void main(String args[])
   {
    MyThread mt=new MyThread();
    mt.start();
   }
}
```

12. How to implement Runnable interface?

Impelementing Runnable interface: we define a class that implements Runnable interface. The Runnable interface has only one method , run( ), that is to be implemented by the class.

It includes the following steps:

- Declare a class that implements Runnable interface.

```
class MyThread implements Runnable
{


}
```

- Implement the run( ) method.

```
public void run()
{
  //Thread code
}
```

- Create a thread by defining an object that is instantiated from this "runnable" class or create it within that class.

```
MyThread mt=new MyThread();
mt.start();
```

Example:

```
public class MyThread implements Runnable
{
   public void run()
   {
      System.out.println("Hello from a thread!");
   }

   public static void main(String args[])
   {
    MyThread mt=new MyThread();
    mt.start();
   }
}
```

13. What are the two methods by which we may stop threads?

Stopping a thread:

Whenever we want to stop a thread from running further, we may do so by calling its **stop*()*** method. This causes a thread to stop immediately and move it to its **dead** state. It forces the thread to stop abruptly before its completion i.e. it causes premature death. To stop a thread we use the following syntax:

```
mt.stop();
```

Blocking a Thread:

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

- **sleep(t)** // blocked for 't' milliseconds
- **suspend()** // blocked until resume() method is invoked
- **wait()** // blocked until notify () is invoked

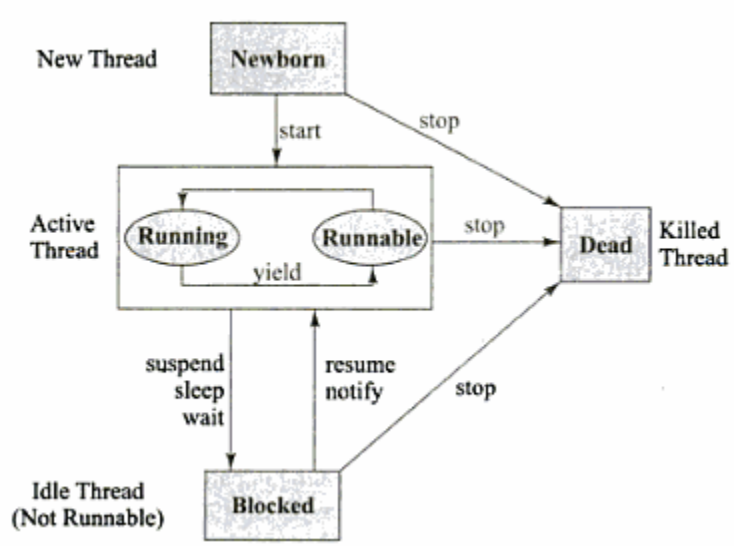
These methods cause the thread to go into the *blocked* (or *not-runnable*) state. The thread will return to the runnable state when the specified time is elapsed in the case of sleep( ), the resume() method is invoked in the case of suspend( ), and the notify( ) method is called in the case of wait().

14. Write about the Life Cycle of Thread.

During the life time of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

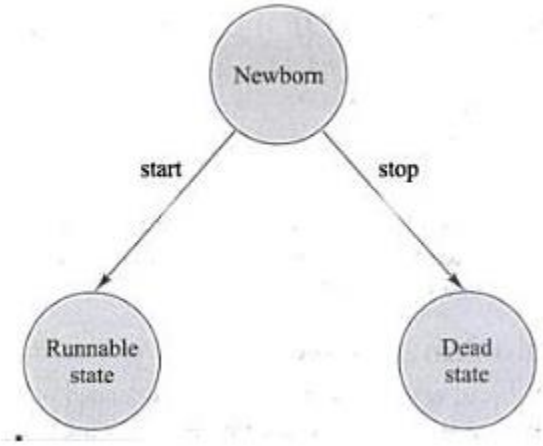
A Thread is always in one of these five states. It can move from one state to another via a variety of ways as follows:



Newborn State: When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

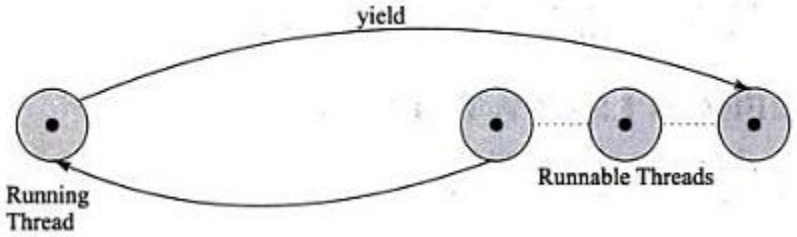
- Schedule it for running using start() method.
- Kill it using stop() method.

If scheduled, it moves to the runnable state If we attempt to use any other method at this stage, an exception will be thrown.



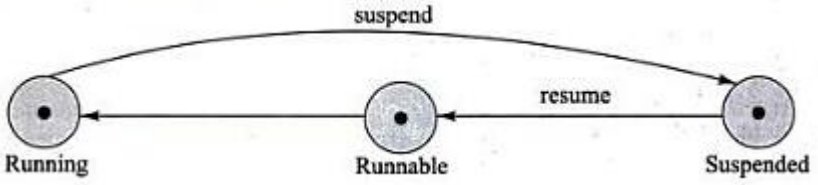
Runnable State: The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the

queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e., first-come, first-serve manner. After its turn, the thread joins the queue again and waits for next turn. This process of assigning time to threads is known as time-slicing. However, if we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using yield() method.

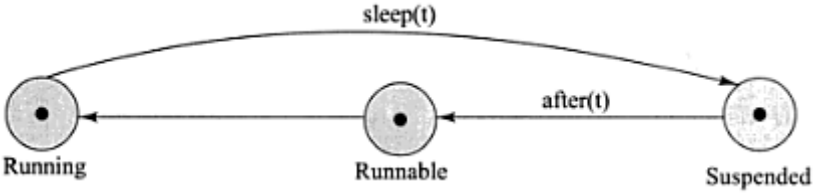


Running State: Running means that the processor has given its time to the thread for its execution. The thread runs until it gives up control on its own or taken over by other threads. When the thread is in its running state, we can ensure that the control is in run() method of the thread.

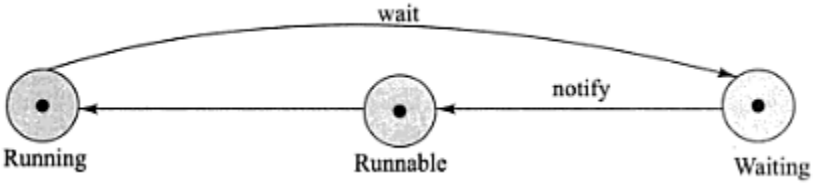
- It has been suspended using **suspend( )** method. A suspended thread can be revived by using the **resume( )** method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.



- It has been made to sleep. We can put a thread to sleep for a specified time period using the method **sleep(time**) where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.



- It has been told to wait until some event occurs. This is done using the **wait( )** method. The thread can be scheduled to run again using the **notify( )** method.



Blocked State: A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

Dead State: Every thread has a life cycle. A running thread ends its life when it has completed executing its **run( )** method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it.

A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.

15. Write a note on various Thread methods.

1. start() : This method is used to start a new thread. When this method is called the thread enters the runnable state and this automatically invokes the run( ) method .

void start( )

2. run(): This method is the most important method in the thread like its heart and soul. It contains the statements that define the actual task of the Thread. It should be overridden in our class in the class of extending Thread class or implement in the case of implementing Runnable interface.

```
void run( )
{
// Thread Code
}
```

3. sleep(): This method is used to block the currently executing thread for the specific time. After the elapse of the time the thread automatically comes into runnable state.

void sleep(long time-in-milliseconds)

4. stop(): This method is used to stop the running thread even before the completion of the task.

void stop()

5. wait() method: This method is used to block the currently executing thread until the thread invokes notify() or notifyall() methods.

void wait()

6. suspend() method: This method is used to block the currently executing thread until the thread invokes resume() method.

void suspend()

7. resume(): This method is used to bring the thread from blocked state to runnable state when it is blocked by suspend() method.

void resume()

8. yield(): This method is used to bring the blocked thread to runnable state. If we want a thread to give a chance to run before its turn comes, we can use the yield() method.

void yield()

9. setPriority(): This method is used to set the priority of the thread. The priority is an integer value that ranges from 1 to 10. The default setting is NORM_PRIORITY whose value is 5. The other constants are: MIN_PRIORITY (= 1) and MAX_ PRIORITY (= 10).

setPriority(int priority)

10. getPriority(): This method is used to get the priority of the thread. It returns integer value.

int getPriority( )

16. How do we set priorities of Thread? (Mar 2010) (Mar 2011)

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a first-come, first-serve basis. Java permits us to set the priority of a thread using the **setPriority()** method as follows:

ThreadName.setPriority(intNumber);

The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5
- MAX_PRIORITY = 10

The intNumber may assume one of these constants or any value between 1 and 10. The default setting is NORM_PRIORITY. Most user-level processes should use NORM_PRIORITY, plus or minus 1. Whenever multiple threads are ready for execution, the Java system chooses the highest priority thread and executes it.

```
class A extends Thread
{
  public void run()
   {
      System.out.println("Thread  A Started");
      for(int i=1;i<=5;i++)
       {
         System.out.println("Thread A -> No = " +i);
       }
     System.out.println("Exit from Thread A ");
  }
}

class B extends Thread
{
  public void run()
   {
      System.out.println("Thread  B Started");
```

```
      for(int i=1;i<=5;i++)
       {
         System.out.println("Thread B -> No = " +i);
       }
     System.out.println("Exit from Thread B ");
  }
}

class C extends Thread
{
  public void run()
   {
      System.out.println("Thread  C Started");
      for(int i=1;i<=5;i++)
       {
         System.out.println("Thread C -> No = " +i);
       }
     System.out.println("Exit from Thread C ");
  }
}

class ThreadPriority
{
  public static void main(String args[])
  {
        A obja=new A();
        B objb=new B();
        C objc=new C();

       objc.setPriority(Thread.MAX_PRIORITY);
       objb.setPriority(obja.getPriority()+1);
       obja.setPriority(Thread.MIN_PRIORITY);

      System.out.println("start thread A");
      obja.start();
      System.out.println("start thread B");
      objb.start();
      System.out.println("start thread C");
      objc.start();
   }
}
```

17. Explain Thread Synchronization.

Threads use their own data and methods provided inside their run() methods. But if we wish to use data and methods outside the thread's run() method, they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as synchronization.

Synchronization can be achieved by two ways:

- Synchronized method
- Synchronized block

Synchronized method: In case of Java, the keyword synchronized helps to solve such problems by keeping a watch on such locations. For example, the method that that will update a file may be declared as synchronized as shown below:

```
synchronized void update( )
{
…….. // code here is synchronized
}
```

Synchronized block: When we declare a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread runs, no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the key can only open the lock.

```
synchronized (lock-object)
{
……. // code here is synchronized
}
```

18. Write about java.io package.

The java.io package in Java provides classes and interfaces for system input and output (I/O) through data streams, serialization, and file handling

The java.io package is largely based on the concept of streams, which represent a continuous flow of data. There are two primary types of streams in Java:

- **Input Streams**: Used to read data (e.g., from a file or network).
  1. FileInputStream
  2. BufferedInputStream

- **Output Streams**: Used to write data (e.g., to a file or console).
  1. FileOutputStream
  2. BufferedOutputStream

- **Byte Streams:** Handle raw binary data (8-bit bytes). Useful when dealing with binary data like images, videos or other media. All Input Streams and Output Streams are examples of Byte Streams
  Examples :InputStream, OutputStream, FileInputStream,FileOutputStream, BufferedInputStream, BufferedOutputStream

19. Write about File Streams.

The two file streams are

- FileInputStream
- FileOutputStream

FileInputStream

FileInputStream class is useful to read data from a file in the form of sequence of bytes. FileInputStream is meant for reading streams of raw bytes such as image data Objects can be created using the keyword new

FileInputStream fis = new FileInputStream("file1.txt");

FileOutputStream

FileOutputStream is an outputstream for writing data/streams of raw bytes to file or storing data to file. FileOutputStream is a subclass of OutputStream

FileOutputStream fos = new FileInputStream("file1.txt");

20. Demonstrate programs for FileInputStream and FileOutputStream.

FileOutputStream:

```
import java.io.*;

public class WriteToFile
{
   public static void main(String[] args)
  {

      try
         {
         FileOutputStream fos = new FileOutputStream("output.txt");
         OutputStreamWriter iow = new OutputStreamWriter(fos);
         BufferedWriter writer = new BufferedWriter(iow);

         String str = "Hello B.Sc.";

         writer.write(str);
         writer.flush();
         writer.close();

       } catch (IOException e) {
          e.printStackTrace();  // Handle the exception
       }
   }
}
```

FileInputStream:

```
import java.io.*;

public class ReadFromFile
{
   public static void main(String[] args)
  {
       try
         {
       FileInputStream fis = new FileInputStream("output.txt");
       InputStreamReader isr = new InputStreamReader(fis);
       BufferedReader reader = new BufferedReader(isr);
```

```
            String s;
            while ((s = reader.readLine()) != null)
            {
               System.out.println(s); // Print each line from the file
            }
            reader.close();
         }
         catch (IOException e)
         {
            e.printStackTrace();  // Handle the exception
         }
      }
   }
```