UNIT–4

EXCEPTION HANDLING & MULTITHREADING

EXCEPTIONS

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exceptions can occur for a variety of reasons, such as division by zero, array index out of bounds, or file not found.For Ex. a user might enter an invalid file-name, or a file might contain corrupted data, a network-link can fail etc.In the solution of these problems, Java supports Exceptions handling. Exception Handling is the mechanism to handle runtime problems. We need to handle such problems to prevent abrupt termination of program. Exceptions can be generated by the Java run-time system or user code also.

When an exceptional event occurs in Java, an exception is said to be thrown. When an exception is occurred, an object, which represents that exception is created and passed to the method where the exception has occurred. This object contains detail information about the exception, which can be retrieved and processed.All exceptions are subclasses of a class called Java.lang.Throwable.

USES OF EXCEPTIONS

In Java, exceptions are used to handle errors and exceptional situations that may occur during the execution of a program. Exceptions provide a way to gracefully handle errors and prevent the program from crashing. Here are some of the main uses of exceptions in Java:

1. Handling Errors: Exceptions are used to handle errors that occur during the execution of a program. For example, if a file cannot be found, an IOException is thrown. By catching and handling this exception, we can notify the user of the error and prevent the program from crashing.
2. Debugging: Exceptions can be used to debug a program by providing more information about the error that occurred. When an exception is thrown, it prints a stack trace that shows where the error occurred, making it easier to identify and fix the problem.
3. Custom Exceptions: In addition to using the built-in exceptions, Java allows us to create custom exceptions to handle specific situations in our program. This can make our code more readable and maintainable.
4. Program Flow Control: Exceptions can be used to control the flow of a program. For example, if an error occurs while processing a record in a database, we can catch the exception and continue processing the next record instead of stopping the program.
5. Separation of Concerns: Exceptions allow us to separate the error handling logic from the main program logic. By catching and handling exceptions in a separate block of code, we can keep the main program logic clean and focused on its primary task.

In summary, exceptions are an essential part of Java programming and are used to handle errors, debug programs, control program flow, and separate concerns. By using exceptions effectively, we can create more robust and maintainable programs.

PURPOSE OF EXCEPTION HANDLING

Exception handling mechanism minimizes the chances of a system crash or abrupt termination of programs.Exception handling mechanism provides more specific information about program errors to user.

EXCEPTION HANDLER

An exception handler is a block of code to handle thrown exception. If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed.

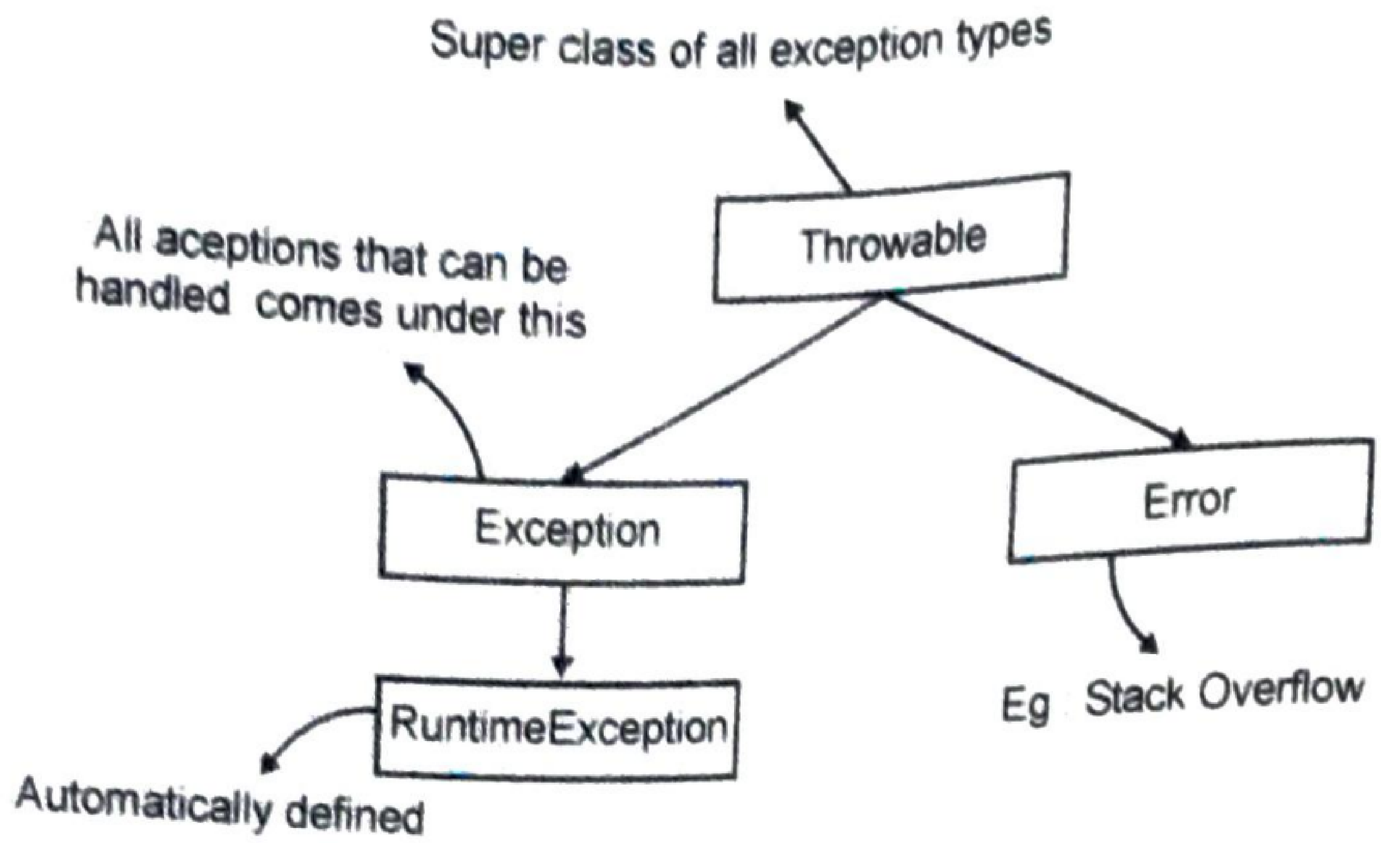
TYPES OF ERRORS

In Java, there are three types of errors: syntax errors, runtime errors, and logical errors.

1. Syntax errors: These are also known as compilation errors. They occur when the Java co detects a problem with the syntax of the code, such as a missing semicolon or ar keyword. Syntax errors prevent the code from being compiled and executed
2. Runtime errors: These occur during the execution of the program. They are c that occur at runtime, such as division by zero, array index out of references. Runtime errors cause the program to terminate abru to identify and fix the problem.

All exception types are subclasses of class Throwable, which is at the top of exception class hierarchy Immediately below Throwable, are two subclasses that partition exceptions into two distinct branches.

1. Exception Class
2. Error Class



EXCEPTION CLASS HIERARCHY

1. Exception class:

This class is used for exceptional conditions that user programs should catch.

This is also the class that you will subclass to create your own custom exception types.

There is an important subclass of Exception, called RuntimeException.

Exceptions of this type are automatically defined for the programs that you write and include things such as division by zeroand invalid array indexing.

2. Error Class:

This class defines exceptions that are not expected to be caught under normal circumstances by your program.

Exceptions of type Errorare used by the Java run-time system to indicate errors having to do with the run-time environment, itself.

Stack overflowis an example of such an error.

EXCEPTION HANDLING MECHANISM

In Java, exception handling is done using five keywords,

1. try
2. catch
3. throw
4. throws
5. finally

Exception handling is done by transferring the execution of a program to an appropriate exception handler when exception occurs.

TRY...CATCH STATEMENT

A try/catch block is placed around the code that might generate an exception.Code within a try/catch block is referred to as protected code.Doing so provides two benefits:

1. First, it allows you to fix the error and
2. Second, it prevents the program from automatically terminating.

The try-catch block is used to handle exceptions. The try block encloses the code that might throw an exception, and the catch block catches and handles the exception if it occurs.

Syntax:

```
try
{
    //body
}
catch ( Exception e )
{
    System.out.println (" display your own error message " );
    //body
}
```

Example: Using Try and catch block

```
public class TryCatchExample
{
    public static void main(String[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int c = a / b; // This line will cause an ArithmeticException
            System.out.println("Result: " + c); // This line will not execute
        } catch (ArithmeticException e)
        {
            System.out.println("Caught an exception: " + e.getMessage());
        }
        System.out.println("End of program"); // This line will execute
    }
}
```

In this example, we create a try block that divides a by b, which will cause an ArithmeticException because b is zero. The catch block catches the exception, and prints a message that includes the exception's message. The code outside the try-catch block will execute regardless of whether an exception is thrown.

Output:

```
Caught an exception: / by zero
End of program
```

Note that you can have multiple catch blocks for different types of exceptions, and you can also have a finally block that executes after the try and catch blocks, regardless of whether an exception is thrown.

MULTIPLE CATCH BLOCKS

When multiple types of exceptions can occur in the try block, it is possible to use multiple catch blocks to handle each exception separately. Each catch block must specify the type of exception it can handle, and the order of the catch blocks is important, as the first catch block that matches the exception type will handle the exception. If the exception type matches with the first catch block it gets caught, if not the exception is passed down to the next catch block.This continues until the exception is caught or falls through all catches.After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

Syntax:

```
try
{
    Statements;
}
catch(Exception e1)
{
    ...
}
catch(Exception e2)
{
    ...
}
```

Example: Multiple Catch blocks

Scoshelpers

```
class Student
{
    public static void main(String[] args)
    {
        try
        {
            int arr[]={1,2,3};
            arr[4]=6/0;
        }
        catch(ArithmeticException ae)
```

```
        {
            System.out.println("divide by zero");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("array index out of bound exception");
        }
    }
}
```

Output:

```
divide by zero
```

THROW AND THROWS KEYWORDS

"throw" keyword

"throw" keyword is used to throw an exception explicitly. "throw" is used to indicate that exception has occurred. The operand of throw is an object of a class, which is derived from class Throwable.

Syntax

```
throw Throwableinstance;
```

Where, Throwableinstance must be an object of typeThrowable or a subclass of Throwable.

Creating Instance of Throwable class.

There are two possible ways to get an instance of class Throwable,

Using a parameter in catch block.

```
exc as new throwable("Exception generated");
throw exc
```

Creating instance with new operator.

```
throw new throwable("Exception generated");
```

Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

Example: Demonstrating throw Keyword

Student.java

```
class Student
{
    static void avg()
    {
        try
        {
            throw new ArithmeticException("throw demo");
        }
```

```
        catch(ArithmeticException e)
        {
            System.out.println("Explicit Exception caught");
        }
    }
    public static void main(String args[])
    {
        avg();
    }
}
```

Output:

```
Explicit Exception caught
```

"throws" keyword

The throws keyword is used to declare that a method might throw a checked exception. Checked exceptions are the exceptions that must be handled or declared by the calling method.

Syntax

```
method_type method_name( ) throws exception1, exception2, .... exception
{
    //body
}
```

Example of "throws" statement

```
import java.io.*;
class Student
{
    public static void main(String args[]) throws IOException
    {
        char g;
        g=(char)System.in.read( );
        if(g=='F'||  g=='f')
            System.out.println ("Student is Female");
        else
            System.out.println ("Student is Male");
    }
}
```

Output:

```
F
Student is Female
Or
M
Student is Male
```

FINALLY CLAUSE

Using a finally block, lets you run any clean-up type statements that you want to execute, no matter what happens in the protected code.

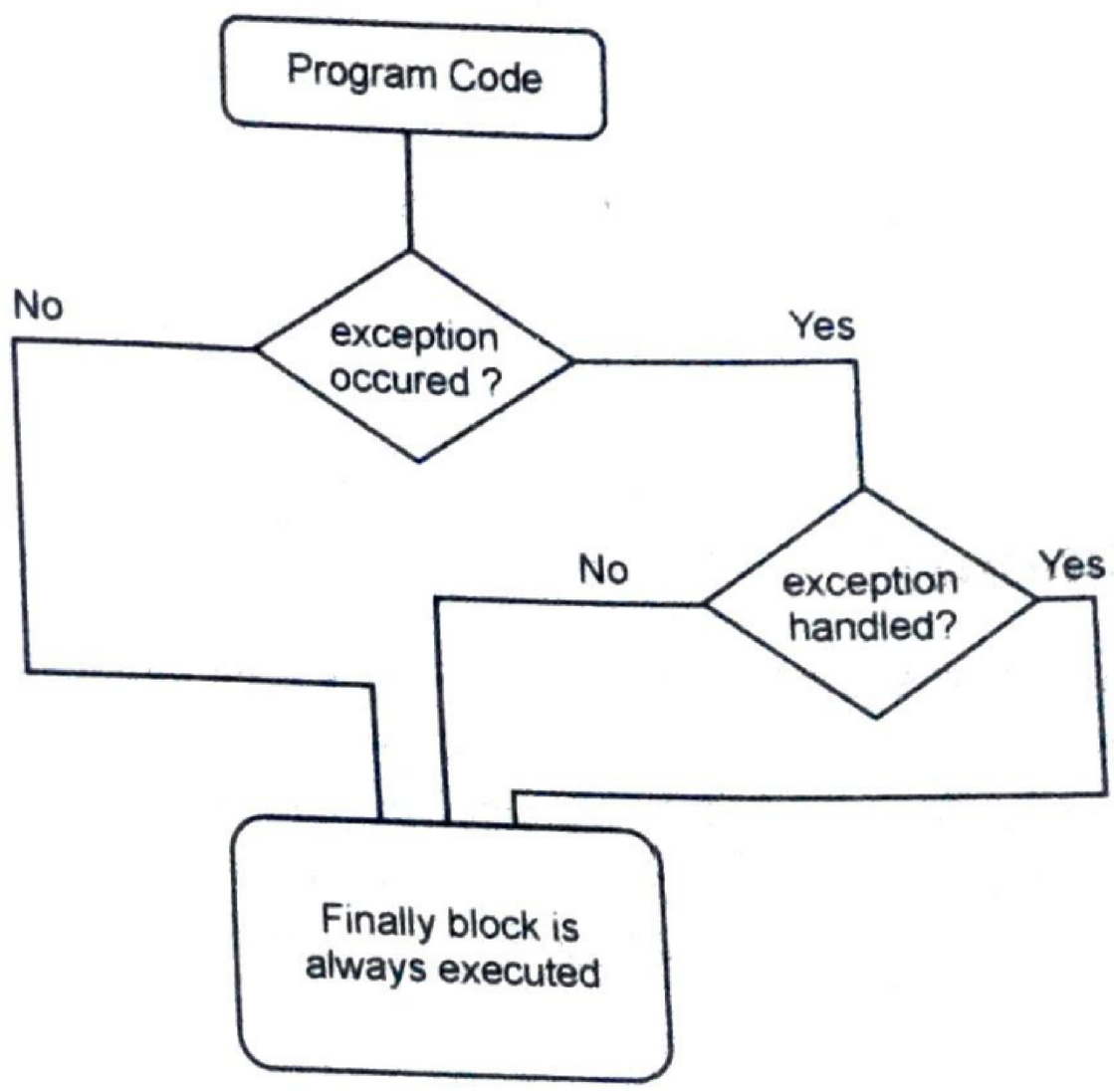
"finally" block is the block in which program return resources to the system and execute other statements for printing any message.

These statements include:

- closing a file
- closing a record set
- closing the connection established with the database
- releasing fonts
- Terminating network connections
- Terminating printer connections etc...

finally" is a keyword that executes a block of statements regardless of whether any exception occurred or not.

- A finally block follows a try block.
- The code of finally block always executes whether or not exception has occurred.



FLOW OF FINALLY BLOCK

Example: Demonstrating finally Clause

```
public class Student
{
    public static void main(String args[ ])
    {
        try
        {
            int a=5;
            int b=0;
            int c= a/b;
        }
        catch (ArithmeticException ae)
        {
            System.out.println ("System message :" + ae.getMessage( ));
            System.out.println ("User message: divided by zero");
        }
        finally
        {
            System.out.println ("this will be always executed");
        }
    }
}
```

Output

```
System message : /  by zero
User message : divided by zero
This will be always executed
```

USER DEFINED EXCEPTIONS

In Java, we can create our own exceptions by extending the Exception class or any of its subclasses. These custom exceptions are called user-defined exceptions. User-defined exceptions can be useful when we want to create an exception that is specific to our program or application.This can be done simply by extending Java Exception class. You can define a constructor for your Exception sub class (not compulsory) and you can override the toString() function to display your customized message on catch.

Example: Custom Exception

```
class Student1 extends Exception
{
    private int rollno;
    Student1(int a)
    {
        rollno=a;
    }
    public String toString()
    {
        return "MyException[" + rollno +"] is less than zero";
    }
}
class Student
{
    static void sum(int a,int b) throws Student1
    {
        if(a<0)
        {
            throw new Student1(a);        //throws user defined exception object
        }
        else
        {
            System.out.println(a+b);
        }
    }
    public static void main(String[] args)
    {
        try
        {
            sum(-10, 10);
        }
        catch(Student1 me)
        {
            System.out.println(me);
        }
    }
}
```

Output

```
MyException[" -10"] is less than zero
```

CONCEPT OF MULTITHREADING

Multithreading in Java refers to the capability of a program to perform multiple tasks concurrently, in other words, to execute multiple threads of execution simultaneously. A thread is a lightweight subprocess, a separate path of execution, that allows a program to perform several tasks concurrently.

Java provides built-in support for multithreading through its java.lang.Thread class and related interfaces. The Thread class provides methods to create, start, stop, and manage threads. A Java program can create multiple threads by extending the Thread class or by implementing the Runnable interface.

When a thread is created, it is assigned a priority, which determines the order in which threads are executed. The higher the priority, the more likely the thread will be executed first.

Java also provides synchronization mechanisms to prevent multiple threads from accessing shared resources at the same time, which could result in data corruption or other problems. Synchronization is achieved through the use of locks and monitors, which ensure that only one thread can access a shared resource at a time.

Multithreading is commonly used in Java to improve the performance of programs that involve input/output operations, user interface updates, and network communications, among others. By allowing multiple threads to execute concurrently, a program can perform several tasks at once, improving its responsiveness and throughput.

Process

A process is an instance of a computer program that is executed sequentially.

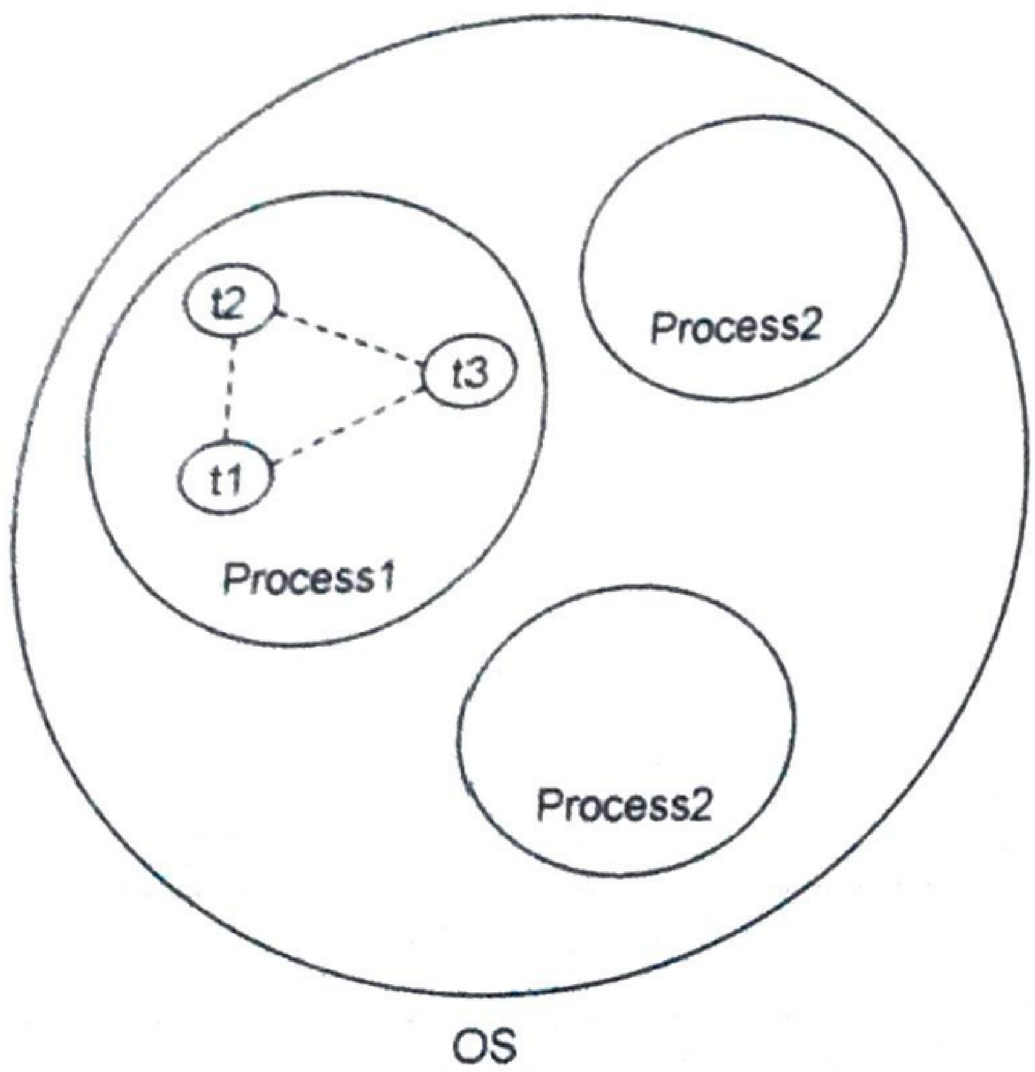
For example, to check the spelling is a single process in the Word Processor program and you can also use other processes like printing, formatting, drawing, etc. associated with this program.

Thread

A thread is a lightweight subprocess, a smallest unit of processing.

It is a separate path of execution.

It shares the memory area of process.



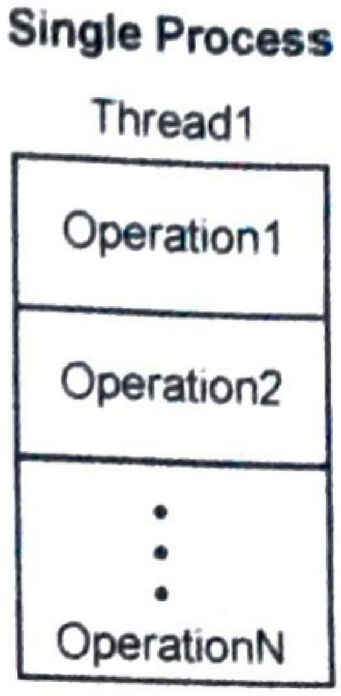
THREAD WITHIN PROCESS

As shown in the figure, thread is executed inside the process.

There can be multiple processes inside the OS and one process can have multiple threads.

Threading concept is very important in Java through which we can increase the speed of any application.

You can see diagram shown below in which a thread is executed along with its several operations within a single process.

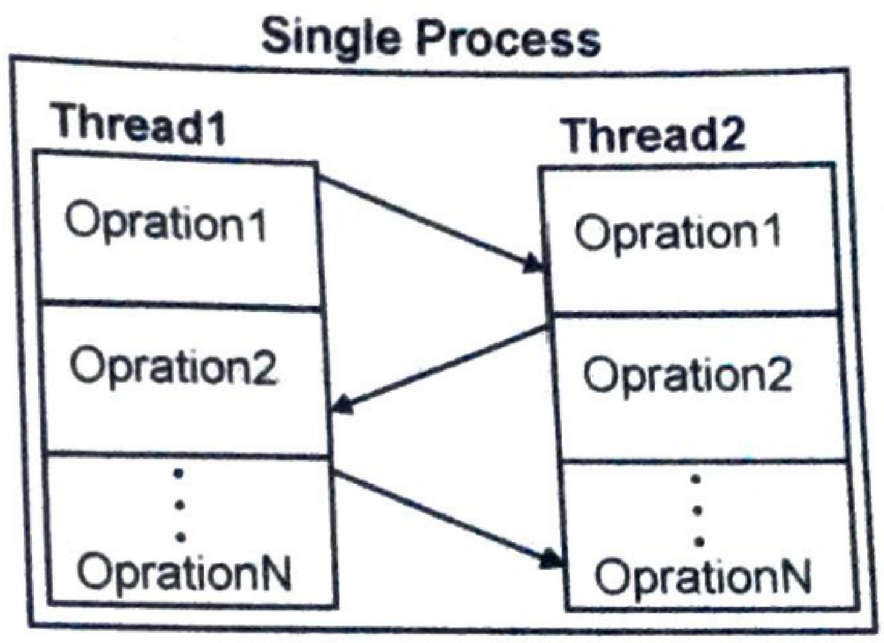


Multithreaded programs contain two or more threads that can run concurrently.

For example, one thread is writing content on a file at the same time another thread is performing spelling check. Several threads of execution may be associated with a single process.

When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.

For example, look at the diagram as shown below:



MULTITHREADING CONCEPTS

In this diagram, two threads are being executed having more than one task.

The task of each thread is switched to the task of another thread.

In Java, multithreaded programming provides facility of

- Creating multiple threads from the same Runnable instance
- Threads share the same data and code.

Example: A single thread creation implementing the Runnable interface

```
class MyThread1 implements Runnable
{
    Thread t;
    String s=null;
    MyThread1(String s1)
    {
        s=s1;
        t=new Thread(this);
        t.start();
    }
    public void run()
    {
        System.out.println(s);
    }
}
public class Student
{
    public static void main(String args[])
    {
        MyThread1 m1=new MyThread1("Thread started....");
    }
}
```

Output:

```
Thread started
```

There are two reasons for implementing a Runnable interface preferable to extending the Thread Class:

If you extend the Thread Class, that means that subclass cannot extend any other Class, but if you implement Runnable interface then you can do this.

The class implementing the Runnable interface can avoid the full overhead of Thread class which can be excessive.

Understand thread execution

- To execute a thread, the thread is first created and then the start() method is invoked on the thread.
- Eventually the thread would execute and the run method would be invoked.
- When a new thread is started, the entry point into the program is the run() method.
- The first statement in run() will be the first statement executed by the new thread.

- Every statement that the thread will execute is included in the run() method or is in other methods invoked directly or indirectly by run().
- The new thread is considered to be alive from just before run() is called until just after run() returns, at which time the thread dies.
- After a thread has died, it cannot be restarted.

CREATING MULTIPLE THREADS IN JAVA

Like creation of a single thread, you can also create more than one thread (multithreads) in a program using class Thread or implementing interface Runnable.

Using the Thread class:

Extend the Thread class and override its run() method with the code you want to execute in the new thread. b. Instantiate the Thread subclass and call its start() method to start the thread.

For example:

```
class MyThread extends Thread
{
    public void run()
    {
        // code to be executed in the new thread
    }
}

// create and start multiple threads
MyThread thread1 = new MyThread();
thread1.start();

MyThread thread2 = new MyThread();
thread2.start();
```

Example: Implementation of the multithreads by extending Thread Class

```
class MyThread extends Thread
{
    MyThread(String s)
    {
        super(s);
        start();
    }
    public void run()
    {
        for(int i=0;i<5;i++)
```

```
        {
            System.out.println("Thread Name  :"
                +Thread.currentThread().getName());
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception e){}
        }
    }
}
public class Student
{
    public static void main(String args[])
    {
        System.out.println("Thread Name :"
            +Thread.currentThread().getName());
        MyThread m1=new MyThread("My Thread 1");
        MyThread m2=new MyThread("My Thread 2");
    }
}
```

Output:

```
Thread Name :main
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
```

In this program, two threads are created along with the "main" thread.The currentThread() method of theThread class returns a reference to the currently executing thread and the getName( ) method returns the name of the thread. The sleep( ) method pauses execution of the current thread for 1000 milliseconds(1 second) and switches to the another threads to execute it. At the time of execution of the program, both threads are registered with the thread scheduler and the CPU scheduler executes them one by one.

Using the Runnable interface:

Implement the Runnable interface and override its run() method with the code you want to execute in the new thread. b. Instantiate a Thread object and pass the Runnable implementation as a parameter to its constructor. c. Call the start() method of the Thread object to start the thread.

For example:

```
class MyRunnable implements Runnable
{
    public void run()
    {
        // code to be executed in the new thread
    }
}

// create and start multiple threads
MyRunnable runnable1 = new MyRunnable();
Thread thread1 = new Thread(runnable1);
thread1.start();

MyRunnable runnable2 = new MyRunnable();
Thread thread2 = new Thread(runnable2);
thread2.start();
```

Both approaches will create multiple threads that can run concurrently with the main thread. You can also use synchronization techniques such as locks or semaphores to coordinate access to shared resources among the threads.

Example: Implementation of the multithreads by implementing the runnable interface

```
class MyThread1 implements Runnable
{
    Thread t;
    MyThread1(String s)
    {
        t=new Thread(this,s);
        t.start();
    }
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("Thread Name  :"+Thread.currentThread().getName());
            try
            {
                Thread.sleep(1000);
            }catch(Exception e){}
        }
    }
}
```

```
public class Student
{
    public static void main(String args[])
    {
        System.out.println("Thread Name :"+Thread.currentThread().getName());
        MyThread1 m1=new MyThread1("My Thread 1");
        MyThread1 m2=new MyThread1("My Thread 2");
    }
}
```

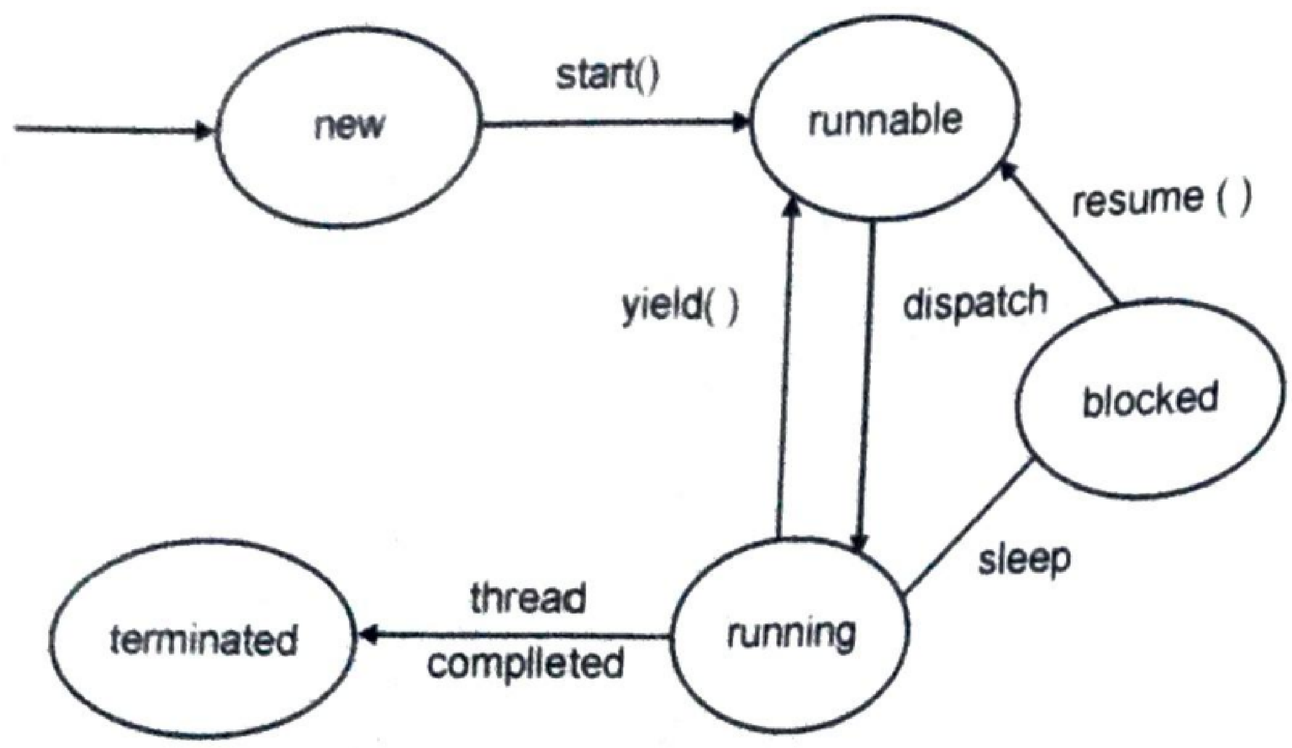
Output:

```
Thread Name :main
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
```

Note that, this program gives the same output as the output of the previous example, it means, you can use either class Thread or interface Runnable to implement thread in your program.

LIFE CYCLE OF A THREAD

When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states.



LIFE CYCLE OF A THREAD

1. New: A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.

2. Runnable: After invocation of start() method on new thread, the thread becomes runnable.
3. Running: A method is in running thread if the thread scheduler has selected it.
4. Waiting: A thread is waiting for another thread to perform a task. In this stage, the thread is still alive.
5. Terminated: A thread enters the terminated state when it completes its task.

THREAD PRIORITY

Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution.

In Java thread priority ranges between,

- MIN-PRIORITY (a constant of 1)
- MAX-PRIORITY (a constant of 10)

Constant	Description
Thread.MIN_PRIORITY	The maximum priority of any thread (an int value of 10)
Thread.MAX_PRIORITY	The minimum priority of any thread (an int value of 1)
Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

By default, every thread is given a NORM-PRIORITY(5). The main thread always has NORM-PRIORITY.

The methods that are used to set the priority of thread shown as:

Thread Priority methods

Method	Description
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

Example: Demonstration of Thread priorities

```
class display implements Runnable
{
    public void run( )
    {
        int i=0;
        while(i<4)
        System.out.println ("Hello : " + i++);
    }
}
```

```
public class Student
{
    public static void main(String args [ ])
    {
        display d=new display( );
        Thread t1=new Thread(d);
        Thread t2=new Thread(d);
        Thread t3=new Thread(d);
        System.out.println ("current priority of thread t1 is :" + t1.getPriority( ));
        t1.setPriority(3);
        t1.start( );
        t2.start( );
        t3.start( );
        System.out.println ("new priority of thread t1 is :" + t1.getPriority( ));
    }
}
```

Output:

```
current priority of thread t1 is :5
new priority of thread t1 is :3
Hello : 0
Hello : 0
Hello : 1
Hello : 1
Hello : 2
Hello : 2
Hello : 3
Hello : 3
Hello : 0
Hello : 1
Hello : 2
Hello : 3
```

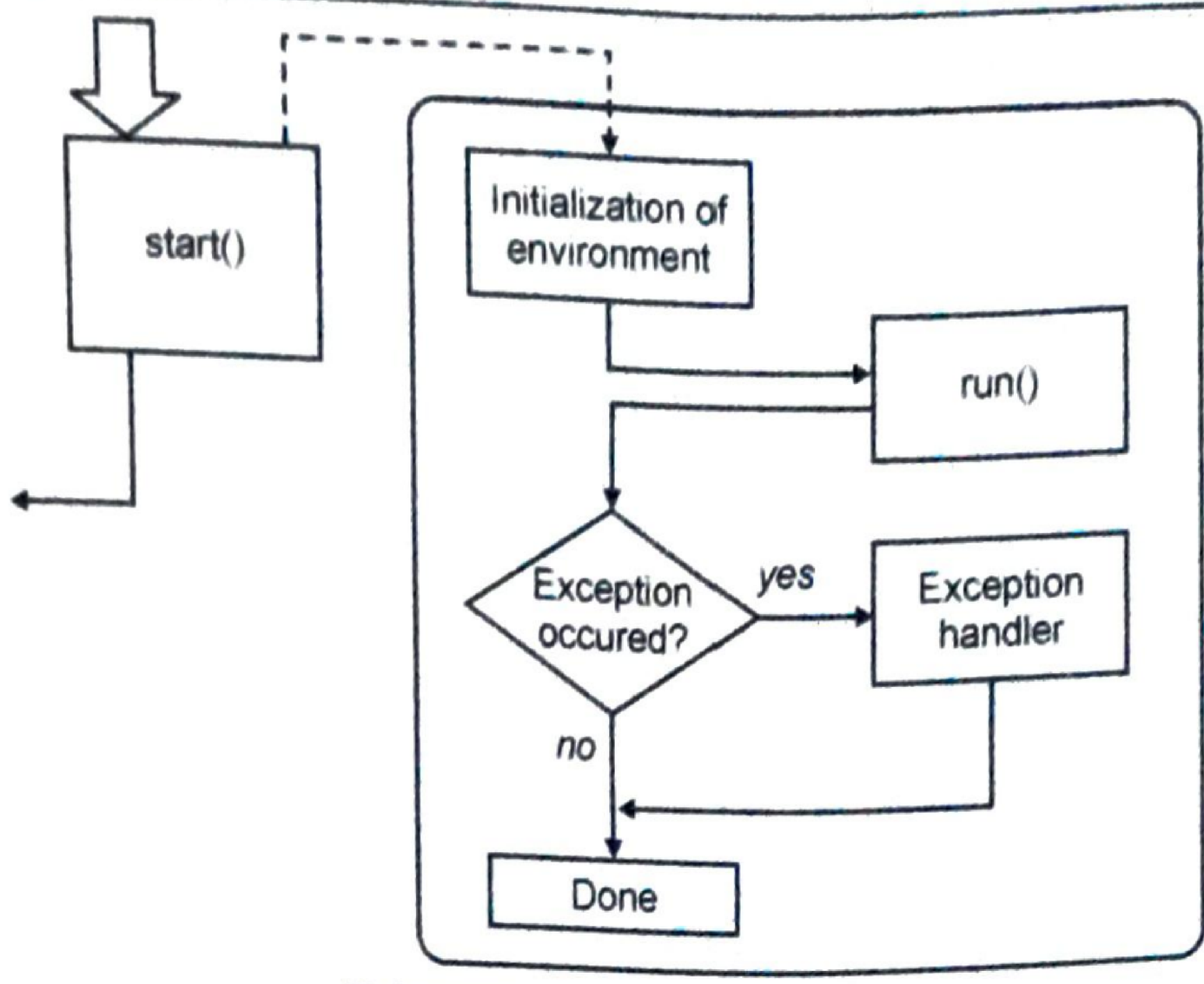
Note the last 4 statements in output they are the execution statements of thread t1 due to low priority.

EXCEPTION HANDLING IN THREADS

When we create a thread the start() method performs some internal housekeeping and then calls the run( ) method.

The start() method does start another thread of control, but the run() method is not really the "main" method of the new thread.

The run()method is executed inside a context that allows the virtual machine to handle runtime exceptions thrown from the run()method. This process is shown in figure 5-10.



EXCEPTION HANDING IN THREADS

All uncaught exceptions are handled by code outside of the run() method before the thread terminates.

The default exception handler is a Java method; it can be overridden; this means that it is possible for a program to write a new default exception handler.

The default exception handler is the uncaughtException() method of the ThreadGroup class, it is called only when an exception is thrown from the run() method.

The thread is technically completed when the run() method returns, even though the exception handler is still running the thread.

Whenever an uncaught exception occurs in a thread's run method, we get a default exception dump which gets printed on System.err stream.

Thread class has two methods:

```
setDefaultUncaughtExceptionHandler()
```

This static method of Thread class can be used to provide a common exception handler for all the threads.

```
setUncaughtExceptionHandler()
```

This non static method of Thread class which is used to provide handler to a given thread.

Example: Demonstration to handle the uncaught exception thrown from a Thread

Student.java

```
public class Student
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new ThreadWithException());
        // Add the handler to the thread object
        t1.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler()
```

```
        {
            public void uncaughtException(Thread t, Throwable e)
            {
                System.out.println("ERROR! An exception occurred in "
                    + t.getName() + ". Cause: " + e.getMessage());
            }
        });
        t1.start();
    }
}
/**
* This thread throws a custom exception in its run method.
*/
class ThreadWithException implements Runnable
{
    public void run()
    {
        throw new RuntimeException("Application Specific Exception!!");
    }
}
```

Output:

```
ERROR! An exception occurred in Thread-0. Cause: Application Specific Exception!!
```

MULTIPLE CHOICE QUESTIONS (MCQs)

1. What is an exception in Java?
   (a) A type of control statement
   (b) An error in code execution
   (c) A special keyword for declaring methods
   (d) A class that represents a value or condition
2. What is the purpose of try-catch block in Java?
   (a) To handle errors in code execution
   (b) To terminate the program when an error occurs
   (c) To declare variables for the entire method
   (d) To create a loop in code execution
3. What is the difference between checked and unchecked exceptions in Java?
   (a) Checked exceptions must be caught or declared, while unchecked exceptions do not
   (b) Checked exceptions are more severe than unchecked exceptions
   (c) Checked exceptions are thrown by the JVM, while unchecked exceptions are thrown by the application
   (d) There is no difference between checked and unchecked exceptions

4. Which keyword is used to throw an exception in Java?
(a) throw (b) try (c) catch (d) finally

5. Which of the following is a checked exception in Java?
(a) NullPointerException
(b) ArrayIndexOutOfBoundsException
(c) ArithmeticException
(d) FileNotFoundException

6. Which of the following is an unchecked exception in Java?
(a) IOException
(b) ClassNotFoundException
(c) RuntimeException
(d) SQLException

7. Which keyword is used to handle multiple exceptions in Java?
(a) catch (b) try (c) finally (d) throw

8. Which of the following is not a type of exception in Java?
(a) Error
(b) RuntimeException
(c) Exception
(d) InterfaceException

9. What is a thread in Java?
(a) A method for running multiple programs at once
(b) A block of code that can be executed concurrently with other blocks
(c) A class that represents a user interface element
(d) A tool for debugging code execution

10. What is the purpose of the synchronized keyword in Java?
(a) To prevent multiple threads from executing a block of code at the same time
(b) To force multiple threads to execute a block of code at the same time
(c) To create a new thread for executing code
(d) To stop the execution of a thread

11. What is a deadlock in Java threading?
(a) A situation where multiple threads are waiting for each other to finish executing
(b) A situation where a thread has finished executing but is not terminated
(c) A situation where a thread is running for too long and causing performance issues
(d) A situation where a thread is terminated before finishing its execution

12. What is a race condition in Java threading?
(a) A situation where multiple threads are accessing and modifying the same data concurrently, resulting in unpredictable outcomes
(b) A situation where a thread is terminated before finishing its execution
(c) A situation where a thread has finished executing but is not terminated
(d) A situation where a thread is running for too long and causing performance issues

13. What is a thread pool in Java?
    (a) A collection of threads that are created and managed by the JVM
    (b) A data structure that allows threads to communicate with each other
    (c) A tool for debugging code execution
    (d) A group of threads that are reused to perform a set of tasks
14. What is a thread local variable in Java?
    (a) A variable that is shared by all threads in a program
    (b) A variable that is used to synchronize multiple threads
    (c) A variable that is accessible only by a single thread
    (d) A variable that is used to limit the number of threads in a program
15. What is a daemon thread in Java?
    (a) A thread that is created and managed by the JVM
    (b) A thread that runs in the background and does not prevent the program from terminating
    (c) A thread that is used to synchronize multiple threads
    (d) A thread that is used to limit the number of threads in a program
16. Which keyword is used to prevent multiple threads from accessing the same block of code concurrently in Java?
    (a) synchronize (b) thread (c) pool (d) daemon
17. What is an exception in Java?
    (a) A syntax error (b) A runtime error (c) A logical error (d) A warning message
18. Which keyword is used to declare an exception in Java?
    (a) try (b) catch (c) throw (d) finally
19. What is the purpose of the "finally" block in a try-catch-finally statement in Java?
    (a) To catch exceptions that were not caught by the "catch" block
    (b) To handle exceptions that occurred in the "catch" block
    (c) To execute code that should always be run, whether or not an exception occurred
    (d) To exit the program when an exception occurs
20. Which of the following is true about multithreading in Java?
    (a) Multithreading is the ability to run multiple programs simultaneously.
    (b) Multithreading is the ability to run multiple threads within a single program simultaneously.
    (c) Multithreading is the ability to run multiple programs within a single thread.
    (d) Multithreading is the ability to run a single program on multiple processors simultaneously.
21. What is a thread in Java?
    (a) A separate program that runs independently of other programs.
    (b) A separate process that runs independently of other processes.
    (c) A separate part of a program that runs independently of other parts of the program.
    (d) A separate part of a program that is dependent on other parts of the program.

22. Which method is used to start a new thread in Java?

(a) run() (b) start() (c) execute() (d) begin()

23. What is the difference between a process and a thread in Java?

(a) A process can have multiple threads, but a thread can only have one process.

(b) A thread can have multiple processes, but a process can only have one thread.

(c) A process is a separate program, while a thread is a part of a program.

(d) A thread is a separate program, while a process is a part of a program.

24. What is an exception in Java?

(a) A syntax error (b) A runtime error (c) A logical error (d) A warning message

25. Which keyword is used to declare an exception in Java?

(a) try (b) catch (c) throw (d) finally

26. What is the purpose of the "finally" block in a try-catch-finally statement in Java?

(a) To catch exceptions that were not caught by the "catch" block

(b) To handle exceptions that occurred in the "catch" block

(c) To execute code that should always be run, whether or not an exception occurred

(d) To exit the program when an exception occurs

27. Which of the following is true about multithreading in Java?

(a) Multithreading is the ability to run multiple programs simultaneously.

(b) Multithreading is the ability to run multiple threads within a single program simultaneously.

(c) Multithreading is the ability to run multiple programs within a single thread.

(d) Multithreading is the ability to run a single program on multiple processors simultaneously.

28. What is a thread in Java?

(a) A separate program that runs independently of other programs.

(b) A separate process that runs independently of other processes.

(c) A separate part of a program that runs independently of other parts of the program.

(d) A separate part of a program that is dependent on other parts of the program.

29. Which method is used to start a new thread in Java?

A) run() (b) start() (c) execute() (d) begin()

30. What is the difference between a process and a thread in Java?

(a) A process can have multiple threads, but a thread can only have one process.

(b) A thread can have multiple processes, but a process can only have one thread.

(c) A process is a separate program, while a thread is a part of a program.

(d) A thread is a separate program, while a process is a part of a program.

Answers to MCQs

1. (b) An error in code execution
2. (a) To handle errors in code execution
3. (a) Checked exceptions must be caught or declared, while unchecked exceptions do not
4. (a) throw
5. (d) FileNotFoundException
6. (c) RuntimeException
7. (b) try
8. (d) InterfaceException
9. (b) A block of code that can be executed concurrently with other blocks
10. (a) To prevent multiple threads from executing a block of code at the same time
11. (a) A situation where multiple threads are waiting for each other to finish executing
12. (a) A situation where multiple threads are accessing and modifying the same data concurrently, resulting in unpredictable outcomes
13. (d) A group of threads that are reused to perform a set of tasks
14. (c) A variable that is accessible only by a single thread
15. (b) A thread that runs in the background and does not prevent the program from terminating
16. (a) synchronize
17. (b) A runtime error
18. (c) throw
19. (c) To execute code that should always be run, whether or not an exception occurred
20. (b) Multithreading is the ability to run multiple threads within a single program simultaneously.
21. (c) A separate part of a program that runs independently of other parts of the program.
22. (b) start()
23. (c) A process is a separate program, while a thread is a part of a program.
24. (b) A runtime error
25. (c) throw
26. (c) To execute code that should always be run, whether or not an exception occurred
27. (b) Multithreading is the ability to run multiple threads within a single program simultaneously.
28. (c) A separate part of a program that runs independently of other parts of the program.
29. (b) start()
30. (c) A process is a separate program, while a thread is a part of a program.

IMPORTANT QUESTIONS

Short Questions 2 Marks

1. List any four inbuilt exceptions.

Ans: There are following inbuilt exceptions:

- Arithmetic Exception
- ArrayIndexOutOfBoundException
- ClassNotFoundException
- IOException etc.

2. List different thread priorities.

Ans: There are following Thread Priorities:

- MIN_PRIORITY (Value: 10)
- NORM_PRIORITY (Value: 5)
- MAX_PRIORITY (Value: 1)

3. List types of error in Java.

Ans: The Types of error is categorized in following classes:

- Exception Class
- Error Class

4. How do we set priorities for thread?

Ans: Using setPriority() method we can set priorities of thread.

5. Compare throw Vs throws Keyword.

Ans:

throw keyword	throws keyword
The throw keyword handover our created exception object to JVM manually.	The throws keyword is used to delegate the responsibility of exception handling to the caller of the method.
Synatx is throw Throwable-instance;	Syntax is return_type method_name (parameter-lis) throws ExceptionClass_list {// body of method}
The throw keyword is followed by exception object.	The throws keyword is followed by the list of exception classes that can occur in the method.
It throw a single exception instance.	It declare multiple exception classes separated by comma.

6. Explain throws keyword.

Ans:

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a throwsclause in the method's declaration.

- Athrows clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

7. Explain throw keyword.

Ans:

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for your program to throw an exception explicitly, using the throw statement.
- Syntax: throw ThrowableInstance;
- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

8. Explain Checked Exception.

Ans:

- The Compiler checks the checked exception.
- Except "RuntimeException" class all the child classes of the class "Exception", and the "Error" class and its child classes are Checked Exception.
- If we do not handle the checked exception, then the compiler objects.
- The Program Doesn't Compile if there is an unhandled chaecked exception in the program code.

9. Explain Finally

Ans:

- When exceptions are thrown, execution in a method takes non-sequential path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- With declaration of finally keyword, finally" clause creates a block of statements that will be executedregardless of whether any exception occurred or not.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

10. Define Thread in Java.

Ans:

- A thread is a lightweight subprocess, a smallest unit ofprocessing.
- It is a separate path of execution. It shares the memory area ofprocess.
- Thread is executed inside the process.
- There can be multiple processesinside the OS and one process can have multiple threads.

11. Explain 'try' and 'catch' statements in exceptional handling of Java language.

Ans:

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself.

- Doing so provides two benefits:1. First, it allows you to fix the error2. Second, it prevents the program from automatically terminating.
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block.
- Immediately following the tryblock, include a catch clause that specifies the exception type that youwish to catch.

12. Explain 'Extending Thread class' in Java.

Ans:

- The easiest way to create a thread is to create a new class that extendsThread class, and then to create an instance (object) of that class.
- The extending class must override the run( ) method, which is the entry point for the new thread.
- The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.
- It must also call start( ) to begin execution of the new thread.

13. State two ways to create a thread.

Ans: There are following two ways to Create Thread:

- extends the Thread class
- implement the Runnable Interface

Questions 3 Marks

1. List and explain different types of errors.
2. Explain basic concept of Exception Handling.
3. Write a program in Java to demonstrate use of synchronization of threads when multiple threads are trying to update common variable.

Ans:

Student.java

```
class Student1
{
    int rollno=0;
    synchronized void increment()
    {
        rollno++;
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
            System.out.println("caught:"+e);
        }
        System.out.println("rollno ="+rollno);
    }
}
```

```
class StudentThread extends Thread
{
    Student1 obj;
    StudentThread(Student1 o)
    {
        obj = o;
    }
    public void run()
    {
        obj.increment();
    }
}
public class Student
{
    public static void main(String[] args)
    {
        Student1 s = new Student1();
        StudentThread s1 = new StudentThread(s);
        s1.start();
        StudentThread s2 = new StudentThread(s);
        s2.start();
        StudentThread s3 = new StudentThread(s);
        s3.start();
    }
}
```

Output

```
rollno = 1
rollno = 2
rollno = 3
```

4. Write a program in Java to develop Banking Application in which user deposits the amount ₹ 5000/- and then start withdrawing of ₹ 2000/-, ₹ 1500/- and it throws exception "Not Sufficient Fund" when user withdraws ₹ 3500/- thereafter.

Ans:

```
class MyException extends Exception
{
    MyException(String msg)
    { super(msg); }
}
```

```
class StudentAccount
{
    double balance;
    double interest;
    StudentAccount() { balance=0; interest=0; }
    public void deposit(double amount)
    {
        balance+=amount;
    }
    public void withdraw(double amount)
    {
        balance=balance-amount;
    }
    public void Addinterest()
    {
        balance=balance+balance*interest;
    }
    public double getBalance() { return balance; }
}
public class Student
{
    public static void main(String args[])
    {
        StudentAccount sa=new StudentAccount();
        sa.deposit(5000);
        sa.withdraw(2000);
        sa.Addinterest();
        sa.withdraw(1500);
        sa.Addinterest();
        System.out.println(sa.getBalance());
        try
        {
            sa.withdraw(3500);
            if(sa.getBalance()<0)
            {
                throw new MyException("Not Sufficient Fund");
            }
        }
        catch(MyException e)
        {
            System.out.println(e.getMessage());
            System.out.println("Transaction was Fail..");
        }
    }
}
```

Output

```
1500.0
Not Sufficient Fund
Transaction was Fail.
```

5. Explain throw & throws with example.
6. Describe the complete life cycle of a thread.
7. Write a program in Java to develop user defined exception for 'Divide by Zero' error.

Ans:

```
class MyException extends Exception
{
    String detail;
    MyException(String s) //constructor
    {
        detail = s;
    }
    public String toString()
    {
        return detail;
    }
}
public class Student
{
    public static void main(String args[])
    {
        try
        {
            throw new MyException("Divided by Zero.");
        }
        catch (MyException e)
        {
            System.out.println("Caught:"+e);
        }
    }
}
```

Output

```
Caught: Divied By Zero.
```

Questions 4 Marks

1. Explain throw and finally in Exception Handling of java language.
2. Compare checked and Unchecked Exception.

Ans:

Checked Exception	Unchecked Exception
The Compiler checks the checked exception.	The Compiler does not checks the Unchecked exception.
Except "RuntimeException" class all the child classes of the class "Exception", and the "Error" class and its child classes are Checked Exception.	"RuntimeException"class and its child classes, are unchecked Exceptions.
If we do not handle the checked exception, then the compiler objects.	Even if we do not handle the unchecked exception, the compiler doesn't object.
The Program doesn't Compile if there is an unhandled chaecked exception in the program code.	The program compiles successfully even if there is and unhandled unchecked exception in the program code.
Examples of checked exceptions are IOException, ClassNotFound, DataAccessException etc.	Examples of unchecked exception is Runtime Exceptions or errors.
This Exception is checked by Compiler.	This Exception are not checked by Compiler.

3. Write a Java program that executes two threads. One thread will print the numbers divisible by 3 and another thread print numbers divisible by 5 between 1 to 50.

Ans:

```
public class Student
{
    int count=1;
    int max=50;
    public void printDivisibleby3()
    {
        synchronized (this)
        {
            while (count<max)
            {
                if(count%3==0)
                {
                    try
                    {
                        System.out.println("number divisible by 3:"+count);
                    }
                    catch(Exception e) { e.printStackTrace(); }
                }
                count++;
                notify();
            }
        }
    }
```

```
public void printDivisbleby5()
{
    try { Thread.sleep(1000); } catch(Exception e1) { e1.printStackTrace(); }
    synchronized(this)
    {
        while(count<max)
        {
            if(count%5==0)
            {
                try
                {
                    System.out.println("number divisible by 5:"+count);
                }
                catch(Exception e) { e.printStackTrace(); }
            }
            count++;
            notify();
        }
    }
}
public static void main(String args[])
{
    Student s=new Student();
    Student s1=new Student();
    Thread t1=new Thread(new Runnable()
    {
        @Override
        public void run() { s.printDivisibleby3();}});
        Thread t2=new Thread(new Runnable()
        {
            @Override
            public void run() { s1.printDivisbleby5();}});
            t1.start();
            t2.start();
            try
            {
                t1.join();
                t2.join();
            } catch(Exception e) { e.printStackTrace(); }
        }
    }
}
```

Output

```
number divisible by 3:3
number divisible by 3:6
number divisible by 3:9
number divisible by 3:12
number divisible by 3:15
number divisible by 3:18
number divisible by 3:21
number divisible by 3:24
number divisible by 3:27
number divisible by 3:30
number divisible by 3:33
number divisible by 3:36
number divisible by 3:39
number divisible by 3:42
number divisible by 3:45
number divisible by 3:48
number divisible by 5:5
number divisible by 5:10
number divisible by 5:15
number divisible by 5:20
number divisible by 5:25
number divisible by 5:30
number divisible by 5:35
number divisible by 5:40
number divisible by 5:45
```

4. Explain Try and Catch with Suitable Example.
5. Explain thread synchronization.
6. Explain Exception Handling.
7. Explain multiple catch blocks with example.
8. Explain thread priorities with suitable example.
9. Explain life cycle of a thread.
10. Explain ArrayIndexOutOfBound Exception in Java with example.

Ans:

ArrayIndexOutOfBound Exception:

- The Class of this exception is java.lang.ArrayIndexOutOfBoundsException.
- This exception occurs when you try to access the array index which does not exist.
- For example, if array is having only 5 elements and we trying to display 7th element then it would throw this example.
- The Program example is given below:

Example:

```
public class Student
{
    public static void main(String args[])
    {
        try
        {
            int a[]=new int[10];
            a[11]=9;
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
        System.out.println("Array index doesn't exist");
        }
    }
}
```

Output

```
Array index doesn't exist
```

Questions 7 Marks

1. Explain Synchronization in Thread.
2. What is Thread? List different methods used to create Thread. Explain Thread life cycle.
3. Write a program that executes two threads. One thread will print the even numbers and another thread will print odd numbers from 1 to 50.

Ans:

```
public class Student
{
    boolean odd;
    int count=1;
    int max=50;
    public void printOdd()
    {
        synchronized (this)
        {
            while (count<max)
            {
                while(!odd)
```

```
            {
                try
                {
                    System.out.println("Odd Waiting:"+count);
                    wait();
                    System.out.println("Notified Odd:"+count);
                }
                catch(Exception e) { e.printStackTrace(); }
            }
            System.out.println("Odd Thread:"+count);
            count++;
            odd=false;
            notify();
        }
    }
}
public void printEven()
{
    try { Thread.sleep(1000); } catch(Exception e1) { e1.printStackTrace(); }
    synchronized(this)
    {
        while(count<max)
        {
            while(odd)
            {
                try
                {
                    System.out.println("Even Waiting:"+count);
                    wait();
                    System.out.println("Notified Even:"+count);
                }
                catch(Exception e) { e.printStackTrace(); }
            }
            System.out.println("Even Thread:"+count);
            count++;
            odd=true;
            notify();
        }
    }
}
```

```
public static void main(String args[])
{
    Student s=new Student();
    s.odd=true;
    Thread t1=new Thread(new Runnable()
    {
        @Override
        public void run() { s.printEven();}});
        Thread t2=new Thread(new Runnable()
        {
            @Override
            public void run() { s.printOdd();}});
            t1.start();
            t2.start();
            try
            {
                t1.join();
                t2.join();
            } catch(Exception e) { e.printStackTrace(); }
        }
    }
```

Output

```
Odd Thread: 1
Odd Waiting: 2
Even Thread: 2
Even Waiting: 3
Notified Odd: 3
Odd Thread: 3
Odd Waiting: 4
Notified even: 4
Even Thread: 4
Even Waiting: 5
Notified odd: 5
Like Wise That Print up to 50 Numbers....
```

Write a Java program that executes three threads. One thread displays "Thread–I" every 2500 millisecond, second thread displays "Thread–II" every 5000 millisecond and third thread displays "Thread–III" every 7500 millisecond.

Ans:

```
class student1 extends Thread
{
    synchronized public void run()
    {
        try
        {
            while(true)
            {
                sleep(2500);
                System.out.println("Thread-I");
            }
        } catch(Exception e) {}
    }
}
class student2 extends Thread
{
    synchronized public void run()
    {
        try
        {
            while(true)
            {
                sleep(5000);
                System.out.println("Thread-II");
            }
        } catch(Exception e) {}
    }
}
class student3 extends Thread
{
    synchronized public void run()
    {
        try
        {
            while(true)
            {
                sleep(7500);
                System.out.println("Thread-III");
            }
        } catch(Exception e) {}
    }
}
```

```
public class Student
{
    public static void main(String args[])
    {
        student1 s1=new student1();
        student2 s2=new student2();
        student3 s3=new student3();
        s1.start();
        s2.start();
        s3.start();
    }
}
```

Output

```
Thread-I
Thread-II
Thread-III
Thread-I
Thread-II
Thread-III
Thread-I
Thread-I
Thread-II
Thread-I
Thread-III
Thread-II
Thread-I
Thread-I
Thread-I
Thread-III
Thread-II
Thread-I
Thread-I
Thread-II
Thread-I
Thread-III
Thread-I
Thread-II
Thread-I
Thread-I
Thread-III
Thread-II
Thread-I
Thread-I
Thread-II
Thread-I
Thread-I
It print up to infinity.....
```

ASSIGNMENTS

Part-1: Exception Handling

1. Write a Java program that throws an exception when attempting to divide by zero. Use a try-catch block to catch the exception and display an error message to the user.
2. Write a Java program that reads in integers from the user until they enter a non-numeric value. Use exception handling to catch any input that is not an integer and display an error message to the user.
3. Write a Java program that reads in a file name from the user and attempts to open the file. Use exception handling to catch any errors that occur when opening the file, and display an error message to the user.

Part-2: Multithreading

4. Write a Java program that creates two threads. The first thread should print out the even numbers from 0 to 100, while the second thread should print out the odd numbers from 1 to 99. Make sure that the output from the two threads is interleaved.
5. Write a Java program that creates a thread that performs a long-running task (e.g. sorting a large list). Use a separate thread to display a progress bar that updates every second, indicating the percentage of the task that has been completed.
6. Write a Java program that creates a producer-consumer system using threads. The producer thread should generate random numbers and add them to a shared buffer, while the consumer thread should remove numbers from the buffer and calculate their square roots. Make sure that the producer and consumer threads do not access the shared buffer at the same time.
7. Create a Java program that prompts the user to enter two numbers, and then divides the first number by the second number.
8. Implement exception handling to handle the following scenarios:
   - If the user enters non-numeric characters, display an error message and prompt the user to enter the numbers again.
   - If the user tries to divide by zero, display an error message and prompt the user to enter the numbers again.
   - If any other exception occurs, display an error message and terminate the program.
9. Modify the program to include a try-catch block that catches any exception thrown during execution of the program. Display

■ ✱ ■