COMPUTER SCIENCE

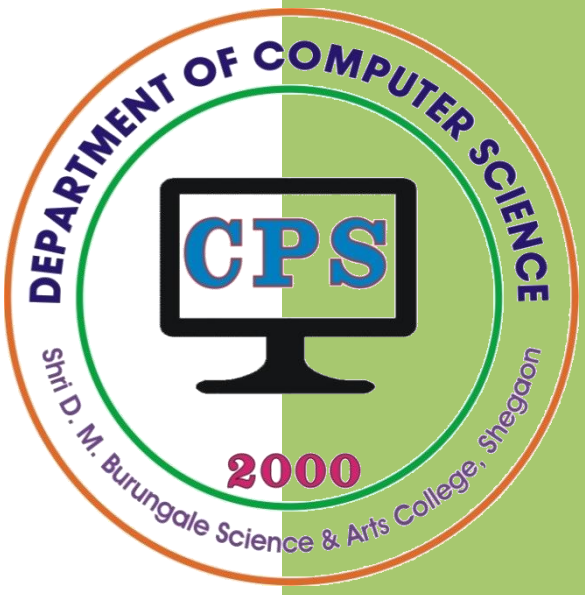
B. Sc. III

Semester-VI

2023-2024

6S : Advanced Java & VB.net

Unit-I : Exception Handling & Multithreading



PROF. V. V. AGARKAR

Assistant Professor & Head

Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

Unit I

Exception Handling and Multithreading: Exception Handling: Concept of Exception handling, Type of Exception, Try, Catch, and Finally, Multiple Catch blocks, Nested Try Statements, throw, throws. **Multithreading:** Multithreading concept, life cycle, creating and running thread, thread priority.

Introduction:

When a new program is created it's rare that it runs successfully at first time. It is common that users make some mistakes while developing or typing program. These mistakes might lead to an error causing to the program to produce unexpected results. Errors are the mistakes or faults in the program. An error may produce an incorrect output or may terminate the execution of the program unexpectedly or even may cause the system to crash. When an error occurs in a program the interpreter will display error massage and immediately stops the execution of program and will not execute the remaining code in a program. Errors may broadly be classified in two categories:

1) Compile-time errors
2) Run-time errors

1) 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 therefore necessary that we fix all the errors before we can successfully compile and run the program. Most of the compile-time errors are due to typing mistakes.

2) 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 or other common run-time errors. These kinds of errors are called as ***run-time errors***. When such errors ate encountered, Java typically generates an error message and aborts the program.

If a program has a run-time error, and instead of terminating the program if you want the program to continue with the execution of the remaining code then it is important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

Exception handling

An *exception* is an unwanted condition that is caused by a run-time error in the program which interrupts the normal flow of the program. There are several reasons that can cause exceptions in a program, for example, opening a non-existing file in your program, network connection problem, and bad input data provided by user etc. When an exception occurs, program execution gets terminated and a system generated error message will display. These messages are not user friendly so a user will not be able to understand what went wrong. Java allows the users to handle the exceptions. Exception handling ensures two things:

1. Meaningful message can be provided to the user about the errors rather than a system generated message, which may not be understandable to a user.
2. The flow of the program doesn't break when an exception occurs.

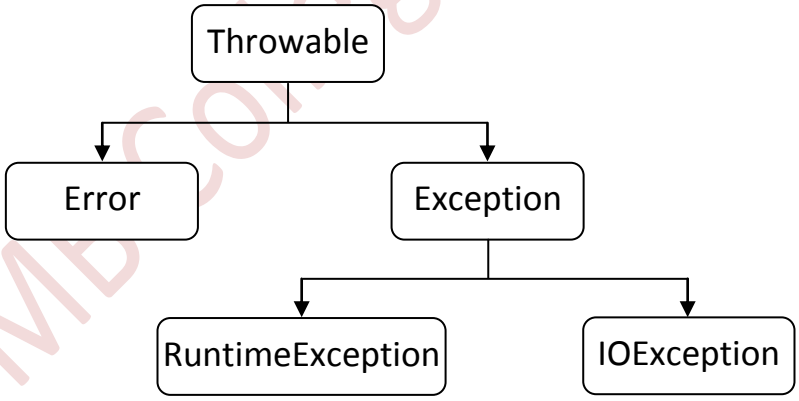
In Java, exception is an object that describes the exception. When inside a method an exception arises, an object representing that exception is created and thrown in the method that caused the error (creating the exception object and handling it to the run-time system is called throwing an exception). That method may choose to handle the exception itself, or pass it on to another method. At some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by users' code.

The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstances" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

1. Find the problem (***Hit*** the exception)
2. Inform that an error has occurred (***Throw*** the exception)
3. Receive the error information (***Catch*** the exception)
4. Take corrective actions (***Handle*** the exception)

The error handling code basically consists of two segments, one to detect errors, and to throw exception and the other to catch exceptions and to take appropriate action.

Types of Exception



[Fig. 1: Types of Exceptions]

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. **Throwable** are two subclasses that partition exceptions into two distinct types: Error and Exception.

1. Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer and we should not try to handle errors.

2. Exceptions

Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred. There are two types of exceptions: RuntimeException and IOException.

i) RuntimeException (unchecked exceptions)

A **runtime exception** happens due to a programming error. They are also known as **unchecked exceptions**. These exceptions are not checked at compile-time but at run-time. Some of the common runtime exceptions are:

- Improper use of an API - `IllegalArgumentException`
- Null pointer access (missing the initialization of a variable) - `NullPointerException`
- Out-of-bounds array access - `ArrayIndexOutOfBoundsException`
- Dividing a number by 0 - `ArithmeticException`

ii) IOException (checked exception)

An IOException is also known as a **checked exception**. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions. Some of the examples of checked exceptions are:

- Trying to open a file that doesn't exist results in `FileNotFoundException`
- Trying to read past the end of a file.

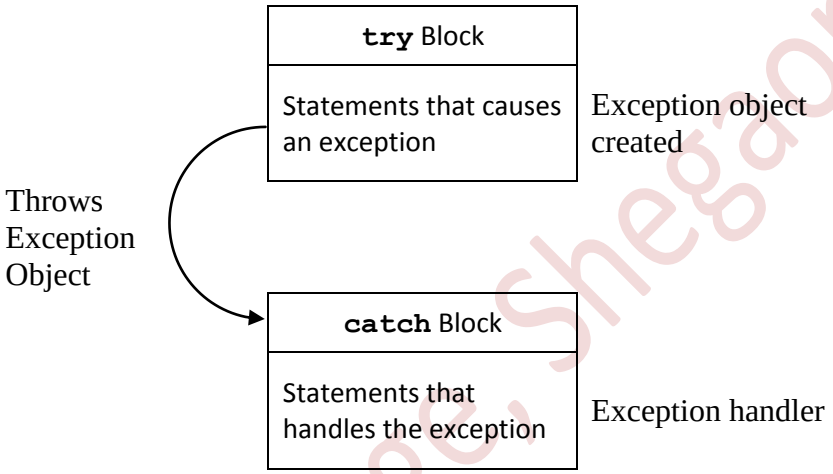
There are some common exceptions that are listed in following table:

Exception Type	*Cause of Exception*
`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 numbers 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 settings
`StackOverFlowException`	Caused when a system runs out of stack space
`StringIndexOutOFBoundsException`	Caused when a program attempts to access a nonexistent character position in a string

Exception Handling Mechanism

Java exception handling is managed via five keywords: `try`, `catch`, `throw`, `throws`, and `finally`. Program statements that want to monitor for exceptions are contained within a `try` block. If an exception occurs within the `try` block, it is thrown. A piece of code can catch this exception (using `catch`) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword `throw`. Any exception that is thrown out of a method must be specified as such by a `throws` clause. Any code that absolutely must be executed after a `try` block completes is put in a `finally` block.

The basic concepts of exception handling are throwing an exception and catching it. This is illustrated in Fig. 2.



[Fig. 2: Exception handling mechanism]

```
try
{
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
finally
{
    // block of code to be executed after try block ends
}
```

The exception handling mechanism consists of three blocks:

1) The **try** block
2) The **catch** block
3) The **finally** block

1) The **try** block

To handle a run-time error, the try block can have one or more statements that could generate an exception. If anyone statement generates an exception, an exception object is created and thrown outside the try block, once an exception is thrown, program control transfers out of the try block into the catch block. The remaining statements in the try

block are skipped and execution jumps to the catch block that is placed next to the try block. Hence the remaining statements in the try block are never executed.

The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement.

2) The **catch** block

Immediately following the try block, includes a catch block that contains exception handling statements. The *ExceptionType* is the type of exception that has occurred. Once the catch statement has executed, program control continues with the next line in the program following the entire *try/catch* mechanism.

A `try` and its `catch` statement form a unit. The scope of the `catch` clause is restricted to those statements specified by the immediately preceding `try` statement. A `catch` statement cannot catch an exception thrown by another `try` statement (except in the case of nested `try` statements).

3) The **finally** block

Java supports another statement known as `finally` statement that can be used to handle an exception that is not caught by any of the previous catch statements. `finally` block can be used to handle any exception generated within a `try` block. It may be added immediately after the `try` block or after the last `catch` block.

When a finally `block` is defined, this is guaranteed to execute, regardless of whether or not the exception is thrown. As a result it can be used to perform certain house-keeping operations such as closing files and releasing system resources etc.

Example:

Following program includes a try block and a catch clause that processes the `ArithmeticException` generated by the division-by-zero error:

```
// Java program to demonstrate ArithmeticException
class Exception1
{
     public static void main(String args[])
     {
     int d, a;

     try
     {    // monitor a block of code
          d = 0;
          a = 42 / d;
          System.out.println("Result = " + a);
     }

     catch (ArithmeticException e)
     {    // catch divide-by-zero error
          System.out.println("Can't divide a number by zero.");
     }

     System.out.println("After catch statement.");
     }
}
```

This program generates the following output:

```
Can't divide a number by zero.
After catch statement.
```

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, user can specify two or more `catch` clauses, each catching a different type of exception. When an exception is thrown, each `catch` statement is inspected in order, and the first one whose type matches that of the exception is executed. After one `catch` statement executes, the others are bypassed, and execution continues after the *try/catch* block.

```
try
{
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
    // exception handler for ExceptionType2
}
......
```

When multiple `catch` statements are used, it is important to remember that exception subclasses must come before any of their superclasses. This is because a `catch` statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Example :

Following program includes a try block and two catch clauses that, one processes the `ArithmeticException` generated by the division-by-zero error and another processes the `ArrayIndexOutOfBoundsException` error:

```
// Java Program to demonstrate multiple catch statements.
class MultiCatch
{
    public static void main(String args[])
    {
      try
      {
          int a = args.length;
          System.out.println("a = " + a);
          int b = 42 / a;
          int c[] = { 1 };
          c[42] = 99;
      }
      catch(ArithmeticException e)
      {
          System.out.println("You should not divide a number by zero");
          System.out.println("Java Exception :  "+e);
      }
      catch(ArrayIndexOutOfBoundsException e)
      {
          System.out.println("You cannot access array elements outside of the
                              limit");
          System.out.println("Java Exception :  "+e);
      }
      System.out.println("After try/catch blocks.");
    }
}
```

This program will cause a division-by-zero exception if it is started with no command line arguments, since `a` will equal zero. It will survive the division if you provide a command-line argument, setting `a` to something larger than zero. But it will cause an `ArrayIndexOutOfBoundsException`, since the `int` array `c` has a length of 1, yet the program attempts to assign a value to `c[42]`.

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
You should not divide a number by zero
Java Exception :  java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatch 25
a = 1
You cannot access array elements outside of the limit
Java Exception :  java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.
```

Nested try Statements

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers (the `try` statement) have to be nested. That is, a `try` statement can be inside the block of another `try`.

When a try statement is present in another try statement then it is called the ***nested try statement***. Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes. If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception. The syntax of nested try statement is as follows:

```
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
```

Example:

```
// Nested try block
class NestedTry
{
   public static void main(String args[])
   {
```

```
        try
        {
            int a[] = {3, 4, 6, 5, 9, 1, 7, 2, 8, 0};
            // displaying element at index 8
            System.out.println("Element at index 8 = "+a[8]);
            // another try block
            try
            {
                System.out.println("Division");
                int res = 100/ 0;
            }
            catch (ArithmeticException ex2)
            {
                System.out.println("Sorry! Division by zero isn't
                                    feasible!");
            }
        }
        catch (ArrayIndexOutOfBoundsException ex1)
        {
            System.out.println("ArrayIndexOutOfBoundsException");
        }
    }
}
```

throw

It is possible in user's program to throw an exception explicitly, using the `throw` keyword. User can define his own set of conditions or rules and throw an exception explicitly. For example, user can throw `ArithmeticException` when a number is divided by 5, or any other numbers. The `throw` keyword explicitly throws an exception from a method or any block of code. User can throw either *checked* or *unchecked* exception. The `throw` keyword is mainly used to throw custom exceptions. The `throw` statement can only throw one exception at a time. The general form of `throw` is :

```
throw ThrowableInstance;
```

or

```
throw new exception_class("error message");
```

Here, *ThrowableInstance* must be an object of type `Throwable` or a subclass of `Throwable`. Primitive types, such as `int` or `char`, as well as non-`Throwable` classes, such as `String` and `Object`, cannot be used as exceptions. There are two ways you can obtain a `Throwable` object: using a parameter in a `catch` clause, or creating one with the `new` operator.

The flow of execution of the program stops immediately after the `throw` statement is executed and the nearest enclosing `try` block is checked to see if it has a `catch` statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing `try` block is checked and so on. If no matching `catch` is found then the default exception handler will halt the program.

Example:

1) `throw new IOException("sorry device error");`
2) `throw new ArithmeticException("not valid");`

throws

If a method can cause an exception that it does not handle, then it must specify this to the callers so that callers can protect themselves against that exception. This can be done by including a `throws` clause in the method's declaration. A `throws` 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. All other exceptions that a method can throw must be declared in the `throws` clause. If they are not, a compile-time error will result.

The general form of a method declaration that includes a throws clause is:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Example:

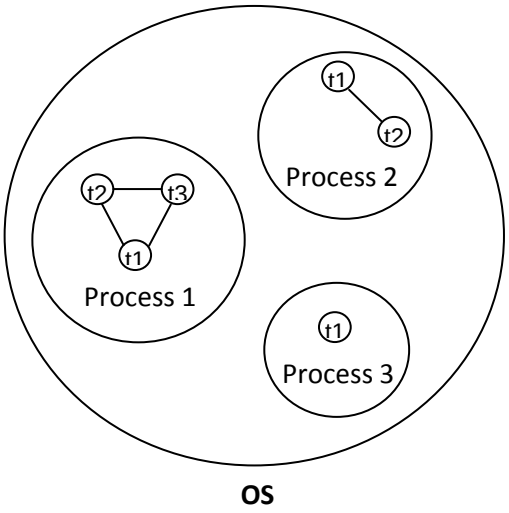
```
// Example of throws
class ThrowsDemo
{
   static void throwOne() throws IllegalAccessException
   {
         System.out.println("Inside throwOne.");
         throw new IllegalAccessException("demo");
   }
   public static void main(String args[])
   {
         try
         {
               throwOne();
         }
         catch (IllegalAccessException e)
         {
               System.out.println("Caught " + e);
         }
   }
}
```

Multithreading

Java provides built-in support for *multithreaded programming.* Multithreading feature of Java allows developing multi-threaded program. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making maximum utilization of the available resources specially CPU. Each part of such program is called a ***thread***. A thread is a lightweight sub-process within a process; it is the smallest unit of processing.

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. Multiprocessing and multithreading, both are used to achieve *multitasking*. Multithreading enables user to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

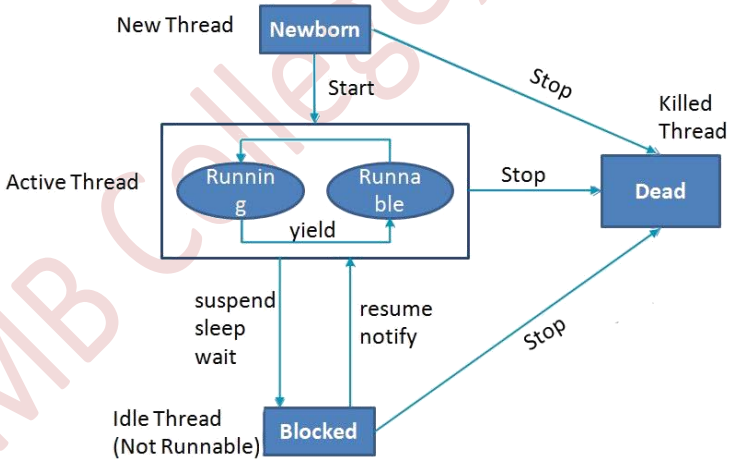
Threads are represented by the `Thread` class and the `Runnable` interface, which are both part of the `java.lang` package of classes. Because they belong to this package, users don't have to use an import statement to make them available in your programs.



[Fig. 3: Process and Thread]

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



[Fig.4 : State transition diagram of a thread]

A thread can be in one of the five states. A thread in Java at any point of time exists in any one of the following states and it can be move from one state to another by different methods and ways.

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

1. Newborn State

When a new thread (thread object) is created, the thread is born and is said to be in *newborn* state. The thread has not yet started to run (started to execute). At this state, user 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 any other method is attempted at this stage, an exception is thrown.

2. Runnable State

The `runnable` stale means that the thread is ready for execution and is waiting for the availability of the processor. That is, thread has joined the queue of threads that are wailing 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. The thread that relinquishes (surrenders) control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as *time-slicing*.

However, if user want a thread to relinquish (surrender) control to another thread to equal priority before its turn comes, this can be done using the `yield()` method.

3. Running State

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

1) It has been suspended using `suspend()` method. A suspended thread can be revived by using the `resume()` method. This approach is useful when user want to suspend a thread for some time due to certain reason, bur do not want to kill it.
2) It has been made to sleep. User can put a thread to sleep for a specified time period using the method `sleep(`*time*`)` where *time* is in millisecond. 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.
3) 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.

4. 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 requirement. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

5. 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, user 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.

Creating and Running thread

Threads can be created by using two mechanisms:

1. Implementing the `Runnable` Interface
2. Extending the `Thread` class

1. Implementing `Runnable` Interface

If user's class is intended to be executed as a thread then this can be achieved by implementing `Runnable` interface. To create threads by implementing `Runnable` interface needs to follow four basic steps:

Step 1: Create a class that `implements` the interface `Runnable` and override `run()` method:

```
class MyThread implements Runnable
{
     ...
     public void run()
     {
          // thread body of execution
     }
}
```

Inside `run()`, you will define the code that constitutes the new thread. The `run()` can call other methods, use other classes, and declare variables, just like the main thread can. 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.

Step 2: Creating Object of a class which implements `Runnable`:

```
MyThread myObject = new MyThread();
```

Step 3: At third step you will instantiate a Thread object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, `threadObj` is an instance of a class that implements the `Runnable` interface and `threadName` is the name given to the new thread.

Step 4: Once Thread object is created, you can start it by calling `start` method, which executes a call to `run` method. Following is simple syntax of start method:

```
Threadobj.start();
```

Example:

```
class MyThread implements Runnable
{
      public void run()
      {
            System.out.println("this thread is running ...");
      }
}
class ThreadEx1
{
      public static void main(String [] args)
```

```
        {
            MyThread myobject = new MyThread();
            Thread t = new Thread(myobject);
            t.start();
        }
    }
```

The class `MyThread` implements standard `Runnable` interface and overrides the `run()` method and includes logic associated with the body of the thread (step 1). The objects created by instantiating the class `MyThread` are normal objects (unlike the first mechanism) (step 2). Therefore, we need to create a generic Thread object and pass `MyThread` object as a parameter to this generic object (step 3). As a result of this association, threaded object is created. In order to execute this threaded object, we need to invoke its `start()` method which sets execution of the new thread (step 4).

2. Extending the `Thread` class

The second way to create a thread is to create a new class that extends **`Thread`** class using the following three simple steps. This approach provides more flexibility in handling multiple threads created using available methods in `Thread` class.

Step 1: Create a class by extending the `Thread` class and override the `run()` method:

```
class MyThread extends Thread {
    public void run() {
        // thread body of execution
    }
}
```

Step 2: Create a thread object:

```
MyThread thr1 = new MyThread();
```

Step 3: Start Execution of created thread:

```
thr1.start();
```

The class `MyThread` extends the standard `Thread` class to gain thread properties through inheritance. The user needs to implement their logic associated with the thread in the `run()` method, which is the body of thread. The objects created by instantiating the class `MyThread` are called threaded objects. Even though the execution method of thread is called run, we do not need to explicitly invoke this method directly. When the `start()` method of a threaded object is invoked, it sets the concurrent execution of the object from that point onward along with the execution of its parent thread/method.

Thread Class *versus* Runnable Interface

By *extending the thread class*, the derived class itself is a thread object and it gains full control over the thread life cycle. *Implementing the Runnable interface* does not give developers any control over the thread itself, as it simply defines the unit of work that will be executed in a thread. Another important point is that when extending the Thread class, the derived class cannot extend any other base classes because Java only allows single inheritance. By implementing the Runnable interface, the class can still extend other base classes if necessary. If the program needs a full control over the thread life cycle, extending the Thread class is a good choice, and if the program needs more flexibility of

extending other base classes, implementing the Runnable interface would be preferable. If none of these is present, either of them is fine to use.

Thread Priority

Whenever a thread is created in Java, it always has some priority assigned to it, that is, it inherits its priority from the thread that created it. Therefore, every thread in Java has a priority. In a Multithreading environment, thread scheduler assigns processor to a thread based on priority of thread. Priorities help the thread scheduler to determine the order in which threads scheduled. The threads with higher priority will usually run before and more frequently than lower priority threads. By default, all the threads has the same priority, i.e., they regarded as being equally distinguished by the scheduler, However, user can explicitly set a thread's priority at any time after its creation by calling its `setPriority()` method. This method accepts an argument of type `int` that defines the new priority of the thread.

Basically, priorities for a thread are represented by an integer value between 1 and 10, with 10 being the highest priority, 1 being the lowest and 5 being the default. There are three static variables defined in `Thread` class for priority.

1. **`public static int MIN_PRIORITY:`** It is the minimum priority of a thread. The value of it is 1.
2. **`public static int NORM_PRIORITY:`** It is the normal priority of a thread. The value of it is 5.
3. **`public static int MAX_PRIORITY:`** It is the maximum priority of a thread. The value of it is 10.

User can also set the priority of thread in between 1 to 10. This priority is known as *custom priority* or *user defined priority*.

The ***default priority*** of a thread is a value '**5**'. User can also determine the current thread priority by calling the **`getPriority`**`()` method.

```
final int getpriority()
```

This method returns an integer value which indicates the current priority of the thread.

For example: To retrieve the current priority of the thread, use the following code.

```
System.out.println("Thread priority ="+t1.getpriority());
```

user can also set a thread's priority by calling the **`setPriority()`** method on a Thread instance.

```
setPriority(int priority)
```

For example: To set the thread to the maximum priority, the following statements should be used.

```
Thread t1 = new thread();
t1.setPriority(Thread.Max_Priority);
```

- **Differences between *throw* and *throws***

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

- **Differences between *errors* and *exceptions***

Errors	Exceptions
1) Impossible to recover from an error	1) Possible to recover from exceptions
2) Errors are of type 'unchecked'	2) Exceptions can be either 'checked' or 'unchecked'
3) Occur at runtime	3) Can occur at compile time or run time
4) Caused by the application running environment	4) Caused by the application itself

■ ■ ■ ■ ■

Sant Gadge Baba Amravati University, Amravati
B. Sc. Part THREE (Semester – VI) Examination
Questions Asked in Previous University Exams

- **Summer-2022 (AY-2272)**

2. A) Explain the life cycle of a thread. 6
   B) Explain nested try statement with suitable example. 6

OR

3. A) How to create multiple threads? Explain with example. 6
   B) Explain various types of exception. 6

- **Winter 2022 (AC-2131)**

2. A) How do we set priorities for threads? Explain. 6
   B) What is multithreading program? Explain how to create the thread. 6

OR

3. A) Explain stopping and blocking a thread with suitable example. 6
   B) Explain throw and throws with suitable example. 6

- **Summer 2023 (AD-1909)**

2. A) What is exception? State and explain need to handle exception. 6
   B) Write procedure to create your own exception. 6

OR

3. A) Explain predefined exception with suitable example. 6
   B) What is thread? Explain thread class with suitable example. 6

- **Winter 2023 (AE-1827)**

2. A) Explain state transition diagram of thread. 6
   B) Explain Finally statement with suitable example. 6

OR

3. A) Explain throw and throws with suitable example. 6
   B) What is multithreading program? Explain how to create thread with example. 6

■ ■ ■ ■ ■