Unit –IV  
Errors and Exceptions: Types of errors, Exception classes, Exception handling in java, use  
of try, catch, finally, throw and throws. Taking user input, Command line arguments.  
Multithreaded Programming: Creating Threads, Life cycle of thread, Thread priority,  
Thread synchronization, Inter-thread communication, Implementing the Runable Interface;

# Types of Errors

Error is an illegal operation performed by the user which results in the abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing.

The most common errors can be broadly classified as follows:

1. **Run Time Error:** Run Time errors occur or we can say, are detected during the execution of the program. Sometimes these are discovered when the user enters an invalid data or data which is not relevant. Runtime errors occur when a program does not contain any syntax errors but asks the computer to do something that the computer is unable to reliably do. During compilation, the compiler has no technique to detect these kinds of errors. It is the JVM (Java Virtual Machine) which detects it while the program is running. To handle the error during the run time we can put our error code inside the try block and catch the error inside the catch block.

**For example:** if the user inputs a data of string format when the computer is expecting an integer, there will be a runtime error.

1. **Compile Time Error:** Compile Time Errors are those errors which prevent the code from running because of an incorrect syntax such as a missing semicolon at the end of a statement or a missing bracket, class not found, etc. These errors are detected by the java compiler and an error message is displayed onto the screen while compiling. Compile Time Errors are sometimes also referred to as **Syntax errors**. These kind of errors are easy to spot and rectify because the java compiler finds them for you. The compiler will tell you which piece of code in the program got in trouble and its best guess as to what you did wrong. Usually, the compiler indicates the exact line where the error is, or sometimes the line just before it, however, if the problem is with incorrectly nested braces, the actual error may be at the beginning of the block. In effect, syntax errors represent grammatical errors in the use of the programming language.

**Example 1:** Misspelled variable name or method names

1. **Logical Error:** A logic error is when your program compiles and executes, but does the wrong thing or returns an incorrect result or no output when it should be returning an output. These errors are detected neither by compiler nor by JVM. The Java system has no idea what your program is supposed to do, so it provides no additional information to help you find the error. Logical errors are also called Semantic Errors. These errors are caused due to an incorrect idea or concept used by a programmer while coding. Syntax errors are grammatical errors whereas, logical errors are errors arising out of an incorrect meaning.

**For example:** if a programmer accidentally adds two variables when he or she meant to divide them, the program will give no error and will execute successfully but with an incorrect result.

**Example:** Accidentally using an incorrect operator on the variables to perform an operation (Using '/' operator to get the modulus instead using '%')

# Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

# Difference between Checked and Unchecked Exceptions

### 1) Checked Exception : The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception : The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error : Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError

# Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

### 1) A scenario where ArithmeticException occurs : If we divide any number by zero, there occurs an ArithmeticException.

**int** a=50/0;//ArithmeticException

### 2) A scenario where NullPointerException occurs : If we have a null value in any [variable](https://www.javatpoint.com/java-variables), performing any operation on the variable throws a NullPointerException.

String s=;

System.out.println(s.length());//NullPointerException

### 3) A scenario where NumberFormatException occurs : The wrong formatting of any value may occur NumberFormatException. Suppose I have a [string](https://www.javatpoint.com/java-string) variable that has characters, converting this variable into digit will occur NumberFormatException.

String s="abc";

**int** i=Integer.parseInt(s);//NumberFormatException

### 4) A scenario where ArrayIndexOutOfBoundsException occurs : If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

**int** a[]=**new** **int**[5];

a[10]=50; //ArrayIndexOutOfBoundsException

**What is an Exception?**

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program’s instructions.

**Error vs Exception**

**Error:**An Error indicates serious problem that a reasonable application should not try to catch.  
**Exception:**Exception indicates conditions that a reasonable application might try to catch.

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

**Advantage of Exception Handling**

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

**Java Exception Keywords**

There are 5 keywords which are used in handling exceptions in Java.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

**Try block**

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

**Catch block**

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

|  |  |
| --- | --- |
| **Try Catch Example** | |
| public class TryCatchExample2  {    public static void main(String[] args)  {  try  {  int data=50/0; //may throw exception  }  //handling the exception  catch(ArithmeticException e)  {  System.out.println(e);  }  System.out.println("rest of the code");  }    } | java.lang.ArithmeticException: / by zero  rest of the code |

**Finally block**

**Java finally block** is a block that is used to execute important code such as closing connection, stream etc.Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

**Why use java finally**

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

|  |  |
| --- | --- |
| **Try Catch Finally Example** | |
| public class TestFinallyBlock2{  public static void main(String args[]){  try{  int data=25/0;  System.out.println(data);  }  catch(ArithmeticException e){System.out.println(e);}  finally{System.out.println("finally block is always executed");}  System.out.println("rest of the code...");  }  } | Output:Exception in thread main java.lang.ArithmeticException:/ by zero  finally block is always executed  rest of the code.. |

**Difference between throw and throws in Java**

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| 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. |

**User Input**

Java **Scanner class** allows the user to take input from the console. It belongs to **java.util** package. It is used to read the input of primitive types like int, double, long, short, float, and byte. It is the easiest way to read input in Java program.

**Syntax**

Scanner sc=**new** Scanner(System.in);

The above statement creates a constructor of the Scanner class having **System.in** as an argument. It means it is going to read from the standard input stream of the program. The **java.util** package should be import while using Scanner class.

It also converts the Bytes (from the input stream) into characters using the platform's default charset.

**Methods of Java Scanner Class**

Java Scanner class provides the following methods to read different primitives types:

|  |  |
| --- | --- |
| **Method** | **Description** |
| **int nextInt()** | It is used to scan the next token of the input as an integer. |
| **float nextFloat()** | It is used to scan the next token of the input as a float. |
| **double nextDouble()** | It is used to scan the next token of the input as a double. |
| **byte nextByte()** | It is used to scan the next token of the input as a byte. |

|  |
| --- |
| **Get Input from User** |
| import java.util.\*;  class UserInputDemo  {  public static void main(String[] args)  {  Scanner sc= new Scanner(System.in); //System.in is a standard input stream  System.out.print("Enter first number- ");  int a= sc.nextInt();  System.out.print("Enter second number- ");  int b= sc.nextInt();  System.out.print("Enter third number- ");  int c= sc.nextInt();  int d=a+b+c;  System.out.println("Total= " +d);  }  } |



**Command Line Arguments**

The java command-line argument is an argument i.e. passed at the time of running the java program.The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

|  |  |
| --- | --- |
| Simple example of command-line argument | Command-line argument that prints all the values |
| 1. **class** CommandLineExample{ 2. **public** **static** **void** main(String args[]){ 3. System.out.println("Your first argument is: "+args[0]); 4. } 5. } 6. compile by > javac CommandLineExample.java 7. run by > java CommandLineExample sonoo   Output: Your first argument is: sonoo | 1. **class** A{ 2. **public** **static** **void** main(String args[]){ 4. **for**(**int** i=0;i<args.length;i++) 5. System.out.println(args[i]); 7. } 8. } 9. compile by > javac A.java 10. run by > java A sonoo jaiswal 1 3 abc   Output: sonoo  jaiswal  1  3  abc |

**Multithreaded Programming**

**Multithreading in java** is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.Java Multithreading is mostly used in games, animation etc.

**Advantages of Java Multithreading**

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

**What is Thread in java**

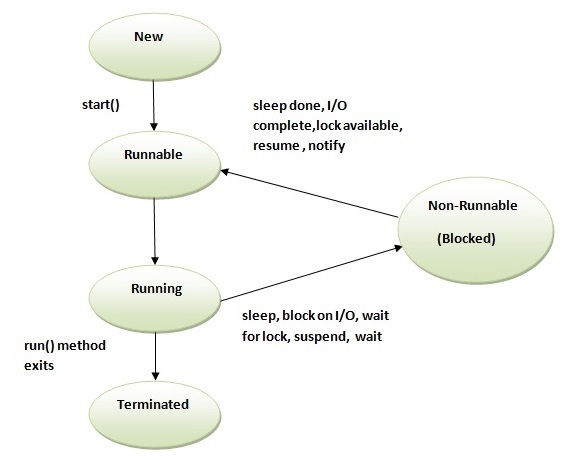
A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

**Life cycle of a Thread**

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java**new, runnable, non-runnable and terminated. There is no running state.But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



|  |
| --- |
| 1) New : The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

### 2) Runnable : The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### 3) Running : The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked) : This is the state when the thread is still alive, but is currently not eligible to run.

### 5) Terminated : A thread is in terminated or dead state when its run() method exits.

**How to create thread**

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

**Thread class:**

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |

**Commonly used methods of Thread class:**

|  |
| --- |
| **public void run():**is used to perform action for a thread.  **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. |
| **public int getPriority():**returns the priority of the thread.  **public int setPriority(int priority):**changes the priority of the thread.  **public void suspend():**is used to suspend the thread(depricated).  **public void resume():**is used to resume the suspended thread(depricated).  **public void stop():**is used to stop the thread(depricated). |

**Runnable interface:**

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

**Starting a thread:**

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

**1) Java Thread Example by extending Thread class**

**class** Multi **extends** Thread{

**public** **void** run(){

System.out.println("thread is running...");

}

**public** **static** **void** main(String args[]){

Multi t1=**new** Multi();

t1.start();

 }

}

**Output:thread is running...**

**2) Java Thread Example by implementing Runnable interface**

**class** Multi3 **implements** Runnable{

**public** **void** run(){

System.out.println("thread is running...");

}

**public** **static** **void** main(String args[]){

Multi3 m1=**new** Multi3();

Thread t1 =**new** Thread(m1);

t1.start();

 }  }

**Thread priority**

|  |
| --- |
| Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. |

**constants defined in Thread class:**

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

|  |
| --- |
| Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10. |

|  |
| --- |
| **Thread Priority** |
| 1. **Class** TestMultiPriority1 **extends** Thread{ 2. **public** **void** run(){ 3. System.out.println("running thread name is:"+Thread.currentThread().getName()); 4. System.out.println("running thread priority is:"+Thread.currentThread().getPriority()); 5. } 6. **public** **static** **void** main(String args[]){ 7. TestMultiPriority1 m1=**new** TestMultiPriority1(); 8. TestMultiPriority1 m2=**new** TestMultiPriority1();   m1.setPriority(Thread.MIN\_PRIORITY);  m2.setPriority(Thread.MAX\_PRIORITY);  m1.start();  m2.start();  }  } |
| Output:running thread name is:Thread-0  running thread priority is:10  running thread name is:Thread-1  running thread priority is:1 |

**Thread synchronization**

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

**Types of Synchronization**

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

**Inter-thread communication in Java**

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

### 1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

### 2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

### 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

**Difference between wait and sleep?**

Let's see the important differences between wait and sleep methods.

|  |  |
| --- | --- |
| wait() | sleep() |
| wait() method releases the lock | sleep() method doesn't release the lock. |
| is the method of Object class | is the method of Thread class |
| is the non-static method | is the static method |
| is the non-static method | is the static method |
| should be notified by notify() or notifyAll() methods | after the specified amount of time, sleep is completed. |