**Session 10**

**OBJECTIVES:** Exception, Checked and unchecked exception, exception hierarchy, try and catch block, multiple catch block, nested catch block, finally block, User-defined exception, throws keyword

**Exceptions**An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

* A user has entered an invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

# Based on these, we have three categories of Exceptions.

**Checked exceptions** − A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

### Example

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

### **Output**

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

**Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

### Example

public class Unchecked\_Demo {

public static void main(String args[]) {

int num[] = {1, 2, 3, 4};

System.out.println(num[5]);

}

}

### **Output**

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)

**Errors**− These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## **Exception Hierarchy**

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Java defines several exception classes inside the standard package java.lang.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

java.lang

Throwable

Exceptions

Error

**Unchecked RuntimeException Defined in java.lang.**

|  |  |
| --- | --- |
| **Sr.No.** | **Exception & Description** |
| 1 | **ArithmeticException**  Arithmetic error, such as divide-by-zero. |
| 2 | **ArrayIndexOutOfBoundsException**  Array index is out-of-bounds. |
| 3 | **ArrayStoreException**  Assignment to an array element of an incompatible type. |
| 4 | **ClassCastException**  Invalid cast. |
| 5 | **IllegalArgumentException**  Illegal argument used to invoke a method. |
| 6 | **IllegalMonitorStateException**  Illegal monitor operation, such as waiting on an unlocked thread. |
| 7 | **IllegalStateException**  Environment or application is in incorrect state. |
| 8 | **IllegalThreadStateException**  Requested operation not compatible with the current thread state. |
| 9 | **IndexOutOfBoundsException**  Some type of index is out-of-bounds. |
| 10 | **NegativeArraySizeException**  Array created with a negative size. |
| 11 | **NullPointerException**  Invalid use of a null reference. |
| 12 | **NumberFormatException**  Invalid conversion of a string to a numeric format. |
| 13 | **SecurityException**  Attempt to violate security. |
| 14 | **StringIndexOutOfBounds**  Attempt to index outside the bounds of a string. |
| 15 | **UnsupportedOperationException**  An unsupported operation was encountered. |

**Checked Exceptions Defined in java.lang.**

|  |  |
| --- | --- |
| **Sr.No.** | **Exception & Description** |
| 1 | **ClassNotFoundException**  Class not found. |
| 2 | **CloneNotSupportedException**  Attempt to clone an object that does not implement the Cloneable interface. |
| 3 | **IllegalAccessException**  Access to a class is denied. |
| 4 | **InstantiationException**  Attempt to create an object of an abstract class or interface. |
| 5 | **InterruptedException**  One thread has been interrupted by another thread. |
| 6 | **NoSuchFieldException**  A requested field does not exist. |
| 7 | **NoSuchMethodException**  A requested method does not exist. |

**try and catch block**

## **Java try block**

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

#### **Syntax of java try-catch:**

try{

//code that may throw exception

}

catch(Exception\_class\_Name ref){}

#### Syntax of try-finally block:

try{

//code that may throw exception

}

finally{}

## **Java catch block**

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

## **Problem without exception handling**

Let's try to understand the problem if we don't use try-catch block.

|  |
| --- |
| 1. public class Testtrycatch1{ 2. public static void main(String args[]){ 3. int data=50/0;//may throw exception 4. System.out.println("rest of the code..."); 5. }   }  Output:  Exception in thread main java.lang.ArithmeticException:/ by zero |

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

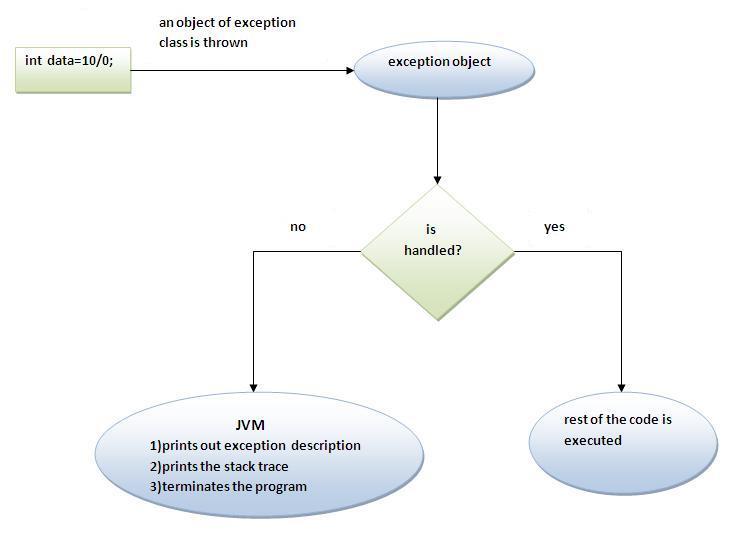
## **Solution by exception handling**

Let's see the solution of above problem by java try-catch block.

|  |
| --- |
| 1. public class Testtrycatch2{ 2. public static void main(String args[]){ 3. try{ 4. int data=50/0; 5. } 6. catch(ArithmeticException e){ 7. System.out.println(e); 8. }   System.out.println("rest of the code...");  }  }  **Output:**  Exception in thread main java.lang.ArithmeticException:/ by zero  rest of the code... |

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

## **Internal working of java try-catch block**



**Multiple catch block**

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block. Let's see a simple example of java multi-catch block.

|  |
| --- |
| 1. public class TestMultipleCatchBlock{ 2. public static void main(String args[]){ 3. try{ 4. int a[]=new int[5]; 5. a[5]=30/0; 6. } 7. catch(ArithmeticException e){System.out.println("task1 is completed");} 8. catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");} 9. catch(Exception e){System.out.println("common task completed");}       System.out.println("rest of the code...");  }  }  Output: task1 completed  rest of the code... |

# **Nested try block**

The try block within a try block is known as nested try block in java.

### **Why use nested try block**

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 have to be nested.

### **Syntax:**

|  |
| --- |
| 1. .... 2. **try** 3. { 4. statement 1; 5. statement 2; 6. **try**   {  statement 1;   1. statement 2;   }  **catch**(Exception e){ }  }  **catch**(Exception e){ }   1. .... |

## **Java nested try example**

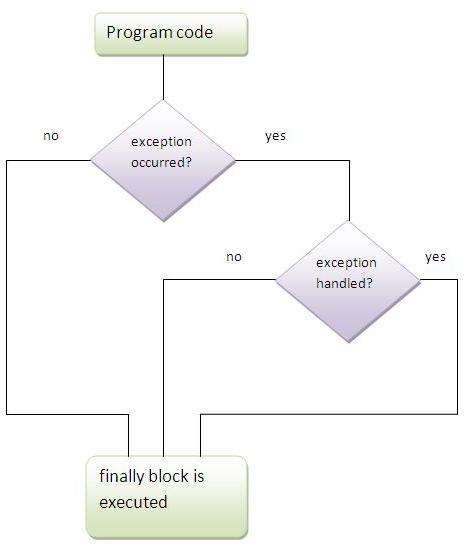
Let's see a simple example of java nested try block.

|  |
| --- |
| 1. class Excep6{ 2. public static void main(String args[]){ 3. try{ 4. try{ 5. System.out.println("going to divide"); 6. int b =39/0; 7. } 8. catch(ArithmeticException e){System.out.println(e);} 9. try{    int a[]=new int[5];       a[5]=4;      }  catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}       System.out.println("other statement);   }  catch(Exception e){System.out.println("handeled");}    System.out.println("normal flow..");  }  } |

**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.

## **throw keyword**

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

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

**throw** ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

**throw** **new** IOException(“demo”);

class TestThrow1{

static void validate(int age){

try{

if(age<18)

throw new ArithmeticException("not valid");

else

System.out.println("welcome to vote");

}

catch(ArithmeticException e){

System.out.println("Exception: "+ e);

}

}

public static void main(String args[]){

validate(33);

System.out.println("rest of the code...");

}

}

# **throws keyword**

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

### **Syntax of java throws**

return\_type method\_name() throws exception\_class\_name{

//method code

}

### **Which exception should be declared?**

 checked exception only, because:

* **unchecked Exception:** under your control so correct your code.
* **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

import java.io.IOException;

class Testthrows1{

void m() throws IOException{

throw new IOException("device error");

}

void n() throws IOException{

m();

}

void p(){

try{

n();

}

catch(IOException e){

System.out.println("exception handled: " +e);

}

}

public static void main(String args[]){

Testthrows1 obj=new Testthrows1();

obj.p();

System.out.println("normal flow...");

}

}

**Exercises:**

1. Write a JAVA code to take input from user and divide 100 by that input. You have to continue this process until the user presses zero. When 100 is divided by zero, print a message that “cannot divide by zero”.
2. Write a JAVA code to take string from user and convert it to integer. If the string cannot be converted to integer, inform the user about this occurrence.
3. Write a Java code segment that will take a sequence of positive integer numbers as input from the keyboard and find the summation of the odd numbers only. If the input is a negative number, your code segment should throw a user-defined exception. The main() method should handle this exception and print the error message.
4. Write a Java program that will take two integer numbers as input from the keyboard. Your program should determine whether the first number is a multiple of the second number.

Your program should provide checking for the following cases:

1. If any of the two numbers is negative.
2. If the first number is smaller than the second number.
3. If the second number is 0.

You should define appropriate exception class for each of the cases and throw an instance of the correct exception when any of the condition arises.