# Exception Handling

## Introduction:

**Bug:** An error in a program is called **bug.**

**Error:** Error is something that most of the time you cannot handle it.

### Types of error in software

* **Syntax error :** Due to the fact that the syntax of the language is not respected.
* **Semantic error :** Due to the fact to an improper use of program statements.
* **Logical Errors :** Due to the fact that the specification is not respected.

### When errors are detected?

* + **Compile Time errors:** Syntax and semantic errors indicated by the compiler.
  + **Runtime errors :** Dynamic semantic errors and logical errors that cannot be detected by the compiler (debugging).

### What is Debugging?

Finding the error from a program is called debugging.

## The Exception Hierarchy:

### What is Exception?

An **exception** is a runtime event which interrupts the normal flow of the program. When an exception occurs, program processing gets terminated and doesn’t continue further.

**Exception** is a class in java

**Exception Hierarchy**

* All exception classes are derived from a class called **java.lang.Exception**, but Exception is a subclass of the

**Throwable** class.

* + So **Throwable** class is the *superclass* of all **errors** and **exceptions** in the Java language.
  + When an exception occurs in a program, an ***object*** of some type of exception class is generated.

### There are mainly three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

**Definitions:**

1. **Checked Exception:**

A checked exception is an exception that occurs at the compile time, these are also called as **compile time exceptions**. These exceptions cannot be ignored at the time of compilation, the programmer should handle these exceptions. Example: IOException, SQLException etc.

### Unchecked Exception:

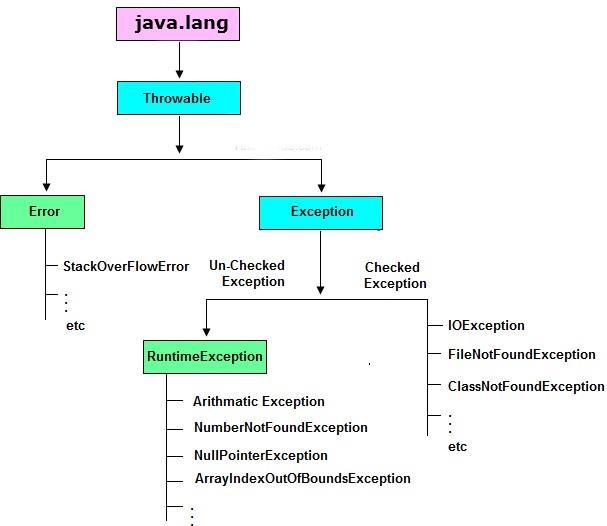
An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

### Error:

Error is **irrecoverable** or can explain 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.

Example: OutOfMemoryError, VirtualMachineError, etc. these are also ignored at the time of compilation.



## Exception Handling Fundamentals:

Java exception handling is managed via 5 keywords. They are

* + **try :** Programs statements that you want to monitor for exceptions
  + **catch :** To handle the Exception.
  + **throw :** To manually throw an exception
  + **throws :** An exception that is *thrown* out of a method
  + **finally :** To *execute important code* while exiting such as *closing* connection, stream etc.

### try {- -}:

* 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 **try{}-catch{}:**

**try {**

**//code that may throw exception**

**}**

**catch(Exception\_class\_Name ref){**

**//code**

**}**

Syntax of **try{}-finally{}:**

**try {**

**//code that may throw exception**

**}**

**finally{**

**//code**

**}**

### catch {- -}:

* + 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.

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

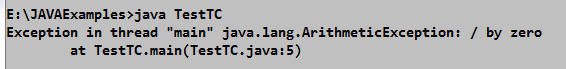
class TestTC{

public static void main(String args[]){ int data=50/0; //may throw exception

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

}

}

Output:

In the above example, when the expression **50/0** is executing it throw the exception hence that rest of the code (In such case rest of the code... statement is not printed) is not executed. If there are 100 lines of code after exception, so all the code after exception will not be executed. Because those exception is handled by JVM standard Exception hence it terminates once the runtime error occurred. Solution for this is use **try** and **catch** block as shown in the below program.

class TestTC{

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

int data=50/0; //may throw exception

}

catch(ArithmeticException e)

{

System.out.println("Error is "+ e);

}

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

}

}

Output:

Now, in the above example, rest of the code is executed i.e. rest of the code... statement is printed. Because the expression **50/0** written within try block hence whenever the exception thrown that catch it in the catch block.

## The Consequences of an Uncaught Exception:

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a

**default exception handler** that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed. This internal working is represented in the below flowchart for the given example.

## Internal working of java try-catch block

**An object exceptio****n class is thrown**

### Exceptions object

**No Is**

### handled?

**Yes**

**JVM**

1. **Prints out exception description**
2. **Prints the stack trace**
3. **Terminates the program**

**Rest of the code is Executed**

It is important to understand that the type of the exception must match the type of specified in a catch block. If it does not, the exception would not be caught. Then such exception is again handled by JVM.



**data = 50/0**

Example:

class TestTC{

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

int data=50/0; //may throw exception

}

catch(**ArrayIndexOutOfBoundsException** e)

{

System.out.println("Error is "+ e);

}

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

}

}

In this program tries to catch an ArithmeticException with a catch for an **ArrayIndexOutOfBoundsException**, but it won’t be caught by catch because it fails to overrun an **ArithmeticException**. Hence JVM will handled.

### Note: Stack trace include the following:

* + the method name, main;
  + the filename; and
  + the line number, 4
  + also include the type of exception thrown is a subclass of Exception called

**ArithmeticException**, which more specifically describes what type of error happened.

## Exceptions Enable you to handle errors gracefully:

One of the key benefits of exception handling is that it enables your program to respond to an error in a graceful, rational way. In some cases, it may be possible to fix the problem and allow the program to continue running.

Example: open a file, network connection, database connection and other task.

class TestTCG

{

public static void main(String args[]){ int[] numer={4,8,16,32,64,128};

int[] denom={2,0,4,4,0,8};

for(int i=0;i<numer.length;i++) try{

System.out.println(numer[i] + "/" + denom[i] +

"=" + **numer[i] / denom[i]**);

}

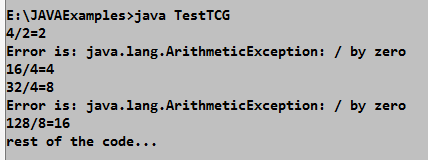
catch(ArithmeticException e){ System.out.println("Error is: " +e);

}

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

}

}

Output:

## Using Multiple catch clauses:

* + If you have to perform different tasks at the occurrence of different Exceptions then you can associate more than one catch clause with a try.
  + Each catch must catch a different type of exception.
  + At a time only one Exception is occurred and at a time only one catch block is executed. Syntax:

**try {**

**//code that may throw exception**

**}**

**catch(Exception\_class\_Name1 ref1){ //code } catch(Exception\_class\_Name2 ref2){ //code }**

class TestTCM

{

public static void main(String args[]){

**int[] numer={4,8,16,32,64,128};**

**int[] denom={2,0,4,4,0};**

for(int i=0;i<numer.length;i++) try{

System.out.println(numer[i] + "/" + denom[i] +

"=" + numer[i] / denom[i]);

}

catch(ArithmeticException e){ System.out.println("Error is: " +e);

}

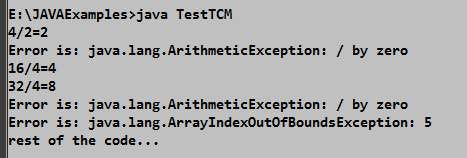
catch(ArrayIndexOutOfBoundsException e1){ System.out.println("Error is: " +e1);

}

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

}

}

Output:

As the output confirms, each catch responds only to its own type of exception.

In general, **catch** clauses are checked in the order in which they occur in a program. Only a matching clause is executed. All other **catch** blocks are ignored.

## Catching subclass Exceptions:

* + There is one important point about multiple catch clauses that relates to **subclasses.**
  + A **catch** clause for a **superclass** will also match any of its **subclasses.**
  + If you want to catch exceptions of both a **superclass** type and a **subclass** type, put the subclass **first** in the **catch** sequence.
  + If you do not, then **superclass catch** will also catch all derived classes.

class TestTCMS{

public static void main(String args[]){ int[] numer={4,8,16,32,64,128,256,512};

int[] denom={2,0,4,4,0};

for(int i=0;i<numer.length;i++) try{

System.out.println(numer[i] + "/" + denom[i] +

"=" + numer[i] / denom[i]);

}

catch(ArithmeticException e){ System.out.println("Error is: " +e);

}

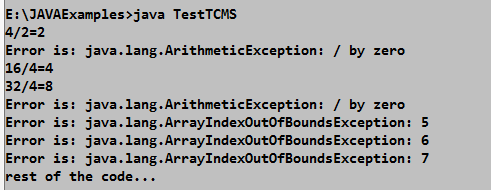
catch(Exception e1){ System.out.println("Error is: " +e1);

}

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

}

}

Output:

In this case, the first catch clause handles **ArithmeticException**. The second one catches all other program- related exceptions, including **ArrayIndexOutofBoundsException** generated when an index value of **denom** is 5, 6, 7 occurs.

The order of catch clauses in the preceding example is important. As an experiment, if you reverse the order of catch clauses like this

catch(Exception e){ System.out.println("Error is: " +e);

}

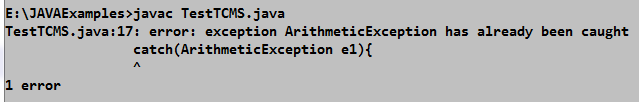
catch(ArithmeticException e1){ System.out.println("Error is: " +e1);

}

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

16MCA32 MODULE-3

Although it **“looks right”**, this sequence won’t compile. As explained, **a subclass exception must be caught before its superclass exception**. Hence, the 1st catch statement will catch all exceptions and the 2nd catch will never be reached, thus producing a compile-time error as show below.



## try blocks can be nested:

* + One try block can be nested within another.
  + An exception generated within the **inner try** block that is not caught by a **catch** associated with that **try**

is propagated to the **outer try** block.

### 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.

Example: Here the **ArrayIndexOutOfBoundsException** is not caught by the **inner catch**, but by the **outer catch**.

class TestTCN {

public static void main(String args[]){ int[] numer={4,8,16,32,64,128,256,512};

int[] denom={2,0,4,4,0,8};

try{ //Outer try

for(int i=0;i<numer.length;i++){ try{ //Inner try

System.out.println(numer[i] + "/" + denom[i] +

"=" + numer[i] / denom[i]);

}//Exit from Inner try catch(ArithmeticException e){

System.out.println("Error is: " +e);

}

}

}//Exit from Outer try catch(ArrayIndexOutOfBoundsException e1){

System.out.println("Error is: " +e1);

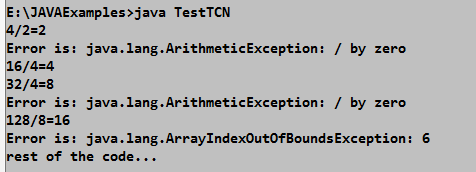
}

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

}

}

By, Suma M G, AP & Rajatha S, AP Dept. of MCA, RNSIT 25 | 34



In this example, an exception that can be handled by the **inner try - -** here, a divide-by-zero error – – allows the program to continue. However, an array boundary error is caught by the outer try, here if it match the catch then allows the program to continue. Otherwise, which causes the program to terminate.

## Throwing an Exception:

* + **“throw”** keyword is used to explicitly throw an exception.
  + We can throw either checked or unchecked exception
  + It is mainly used to throw **custom exception**.

Syntax:



**throw new** Exceptionclass();

**throw new** Exceptionclass(“Any Message”);

Here, **Exceptionclass** is any type of an Exception class (such as custom Exception, checked and unchecked Exception) derived from Throwable.

Example:

class TestThrow{

static void validate(int age){ if(age<18)

**throw new ArithmeticException("not valid");**

else

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

}

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

validate(13);

}

**catch(ArithmeticException ex)**

{

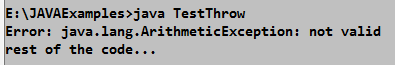
System.out.println("Error: " +ex);

}

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

}

}



In this example, we have created the **validate method** that takes integer value as a parameter. If the **age** is less than 18, we are **throwing** the ***ArithmeticException*,** otherwise print a message **welcome to vote**.

### Rethrowing an Exception:

* When you rethrow an exception, it will not be recaught by the same catch clause.
* It will **propagate** to an **outer catch**.
* To rethrow an exception use Syntax:



**throw** exceptob;

Here, **exceptob** must be an object of an exception class derived from **Throwable**.

Example:

class TestReThrow {

**static void validate(int age){**

try{

if(age<18)

throw new ArithmeticException("not valid");

else

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

}

catch(ArithmeticException ex){ System.out.println("Error is "); **throw ex; //Rethrowing an exception**

}

}

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

**validate(13)**;

}

**//catch rethrown exception**

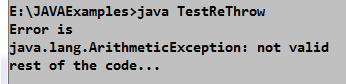
catch(ArithmeticException e){ System.out.println(e);

}

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

}

}



In this program, **ArithmeticException** is handled locally, by **validate()**, but that ArithmeticException is

**rethrown**. Here, that is caught by **main()**.

## A Closer look at Throwable:

All exceptions are subclasses of **Throwable**, all exceptions support the methods defined by **Throwable**. The following is the list of important methods available in the Throwable class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| Throwable **fillInStackTrace()** | Returns a Throwable object that contains a completed stack trace |
| String **getLocalizedMessage()** | Returns a localized description of the exception |
| String **getMessage()** | Returns a description of the exception |
| void **printStackTrace()** | Displays the stack trace |
| void **printStackTrace(PrintStream Stream)** | Sends the stack trace to the specified stream. |
| void **printStackTrace(PrintWriter Stream)** | Sends the stack trace to the specified stream. |
| String **toString()** | Returns a String object containing a complete description of the exception. |

Example:

class TestMethod {

static void validate(int age){ try{

if(age<18)

throw new ArithmeticException("not valid");

else

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

}

catch(ArithmeticException ex){ System.out.println("Error is " +ex ); System.out.println("\nStack Trace: " ); **ex.printStackTrace();** System.out.println(**ex.getMessage()**);

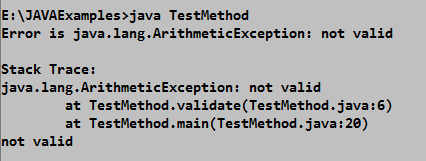
}

}

public static void main(String args[]){ validate(13);

}

}



## using finally:

* + is used *to* ***execute important code*** such as closing connection, stream etc.
  + is always executed whether **exception** is **handled** or **not**.
  + must be followed by **try** or **catch** block.
  + The finally clause is optional. However, each try block there can be **zero** or **more catch blocks**, but **only one finally block**.

Syntax:

**try {**

**//code that may throw exception**

**}**

**catch(Exception\_class\_Name1 ref1){ //code } catch(Exception\_class\_Name2 ref2){ //code } finally{**

**//code**

**}**

Example:

class TestTCF {

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

int data=50/2; //may throw exception

}

catch(ArithmeticException e)

{

System.out.println("Error is "+ e);

}

finally{

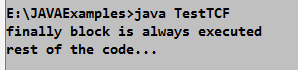
System.out.println("finally block is always executed");

}

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

}

}

Output:

**Difference between final, finally and finalize:**

|  |  |  |
| --- | --- | --- |
| **final** | **finally** | **Finalize** |
| * Final is used to apply restrictions on class, method and variable. * Final class can't be inherited * final method can't be overridden * final variable value can't be changed. | * Finally is used to place important code * it will be executed whether exception is handled or not. | * Finalize is used to perform clean up processing just before object is garbage collected. |
| **“final”** is a keyword. | **“finally**” is a block. | **“finalize”** is a method. |

## using throws:

Sometime method generates an exception that it does not handle, it must declare that exception in a **“throws”**

clause. It is mainly used to handle the **checked** exceptions.

If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he/she is not performing checkup before the code being used.

Syntax:

**Ret-type mehtName(param-list)throws Exce-list{**

**//body**

**}**

Here, **Exce-list** – is a common-separated list of checked exceptions that the method might throw outside of itself Example:

import java.io.\*;

class TestThrows1{

public static void main(String args[])

**throws IOException, Exception**{

int i;

BufferedReader br= new BufferedReader(

new InputStreamReader(System.in));

try{

System.out.println("Enter a number"); i=Integer.parseInt(br.readLine());

**Continued…**

if(i<10)

throw new ArithmeticException();

else

System.out.println("i= " + i);

}

catch(IOException ex){ System.out.println("Some Input error");

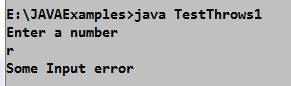
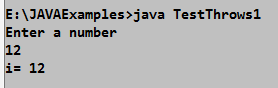
}

catch(Exception ex){ System.out.println("Some Input error");

}

}

}

Output:

**Difference between throw and throws:**

|  |  |
| --- | --- |
| **throw** | **throws** |
| is used to explicitly throw an exception. | is used to declare an exception. |
| Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| is followed by an instance. | is followed by class. |
| is used within the method. | is used with the method signature. |
| You cannot throw multiple exceptions. | You can declare multiple exceptions. Example:  public void method() throws IOException, SQLException. |

## Java’s Built-in Exceptions:

* + Inside the standard package java.lang , Java defines several exception classes.
  + There are two types of exceptions:
    - **checked exceptions:** are checked at compile-time
    - **unchecked exceptions:** are checked at runtime.

### Unchecked Exception:

|  |  |
| --- | --- |
| **Exception** | **Meaning** |
| ArithmeticException | Arithmetic error, such as division by zero |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast |
| IllegalArgumentException | Illegal argument used to invoke a method |
| NullPointerException | Invalid use of a null reference |
| TypeNotPresentException | Type not found |
| NumberFormatException | Invalid conversion of a string to a numeric format |
| StringIndexOutOfBoundsException | Attempt to index outside the bounds of a string |

**Checked Exception:**

|  |  |
| --- | --- |
| **Exception** | **Meaning** |
| ClassNotFoundException | Class not found |
| IllegalAccessException | Access to a class is denied |
| InstantiationException | Attempt to create an object of an abstract class or interface |
| InterruptedException | One thread has been interrupted by another thread |
| NoSuchFieldException | A requested field does not exist |
| NoSuchMethodException | A requested method does not exist |
| ReflectiveOperationException | Superclass of reflection-related exceptions |

## New Exception features added by JDK 7:

There are 3 new features of Exception handling are added in JDK 7. They are:

1. try{ } expanded to **try-with-resources:** also referred as **automatic resource management**, is a new exception handling mechanism, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block.

Following is the **syntax** of **try-with-resources statement**. Syntax:

try**(FileReader fr = new FileReader("file path"))** {

// use the resource

} catch () {

// body of catch

}

}

1. **Multi-catch**: allows two or more exceptions to be caught by the same catch clause as show below. Example:

try{

System.out.println(numer[i] + "/" + denom[i] +

"=" + numer[i] / denom[i]);

}

catch(**ArithmeticException | ArrayIndexOutOfBoundsException** e){ System.out.println("Error is: " +e);

}

### Final rethrow or more precise rethrow:

Refer page No: 28

## Creating Exception Subclasses:

* + Creating an **own exception** is easy.
  + Just define a subclass of **Exception** which is a subclass of **Throwable**.
  + The Exception class does not define any methods of its own. It does inherit those methods provided by

### Throwable.

* + You can override one or more of these methods in exception subclass that you create.
  + Two commonly used Exception constructors are shown here:

### Exception() Exception(String msg);

Example:

class TestUDE{

public static void main(String[] args){ int i=5;

try{

if(i<10)

**throw new MyException("Errorrr");**

else

System.out.println("I= " +i);

}

catch(Exception e){ System.out.println("Message " +e);

}

}

}

//User defined Exeption

**class MyException extends Exception{ MyException(String msg){**

**super(msg);**

**}**

**}**

Output: