# Exception Handling in Java

1. [Exception Handling](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception)
2. [Advantage of Exception Handling](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception#exceptionad)
3. [Hierarchy of Exception classes](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception#exceptionhierarchy)
4. [Types of Exception](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception#exceptiontypes)
   1. [Checked Exception](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception#exceptionchecked)
   2. [Unchecked Exception](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception#exceptionunchecked)
   3. [Error](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception#exceptionerror)
5. [Scenarios where exception may occur](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception#exceptionscenarios)

The exception handling is one of the powerful mechanism provided in java. It provides the mechanism to handle the runtime errors so that normal flow of the application can be maintained.

In this page, we will know about exception, its type and the difference between checked and unchecked exceptions.

### Exception

* **Dictionary Meaning:** Exception is an abnormal condition.
* In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

### Exception Handling

Exception Handling is a mechanism to handle runtime errors.

### Advantage of Exception Handling

|  |
| --- |
| The core advantage of exception handling is that normal flow of the application is maintained. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario: |

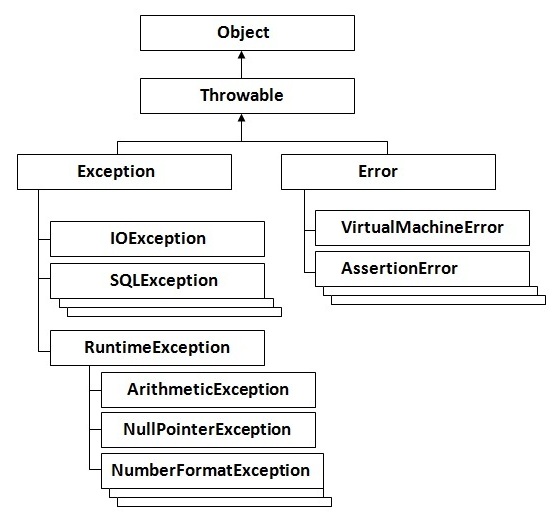
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

|  |
| --- |
| Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the exception will be executed. That is why we use exception handling. |

Do You Know ?

|  |
| --- |
| * What is the difference between checked and unchecked exceptions ? * What happens behind the code int data=50/0; ? * Why use multiple catch block ? * Is there any possibility when finally block is not executed ? * What is exception propagation ? * What is the difference between throw and throws keyword ? * What are the 4 rules for using exception handling with method overriding ? |

### Hierarchy of Exception classes



### Types of Exception:

|  |
| --- |
| There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:   1. Checked Exception 2. Unchecked Exception 3. Error |

### What is the difference between checked and unchecked exceptions ?

### 1)Checked Exception

|  |
| --- |
| The classes that extend 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 that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime. |

### 3)Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

### Common scenarios of Exception Handling where exceptions may occur

|  |
| --- |
| There are given some scenarios where unchecked exceptions can occur. They are as follows: |

### 1) Scenario where ArithmeticException occurs

|  |
| --- |
| If we divide any number by zero, there occurs an ArithmeticException. |

1. int a=50/0;//ArithmeticException

### 2) Scenario where NullPointerException occurs

|  |
| --- |
| If we have null value in any variable, performing any operation by the variable occurs an NullPointerException. |

1. String s=null;
2. System.out.println(s.length());//NullPointerException

### 3) Scenario where NumberFormatException occurs

|  |
| --- |
| The wrong formatting of any value may occur NumberFormatException. Suppose I have string variables that have characters, converting this variable into digit will occur NumberFormatException. |

1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException

### 4) Scenario where ArrayIndexOutOfBoundsException occurs

|  |
| --- |
| If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below: |

1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

# Use of try-catch block in Exception handling:

### Five keywords used in Exception handling:

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

### try block

|  |
| --- |
| Enclose the code that might throw an exception in try block. It must be used within the method and must be followed by either catch or finally block. |

#### Syntax of try with catch block

1. try{
2. ...
3. }catch(Exception\_class\_Name reference){}

#### Syntax of try with finally block

1. try{
2. ...
3. }finally{}

### catch block

|  |
| --- |
| Catch block is used to handle the Exception. It must be used after the try block. |

### Problem without exception handling

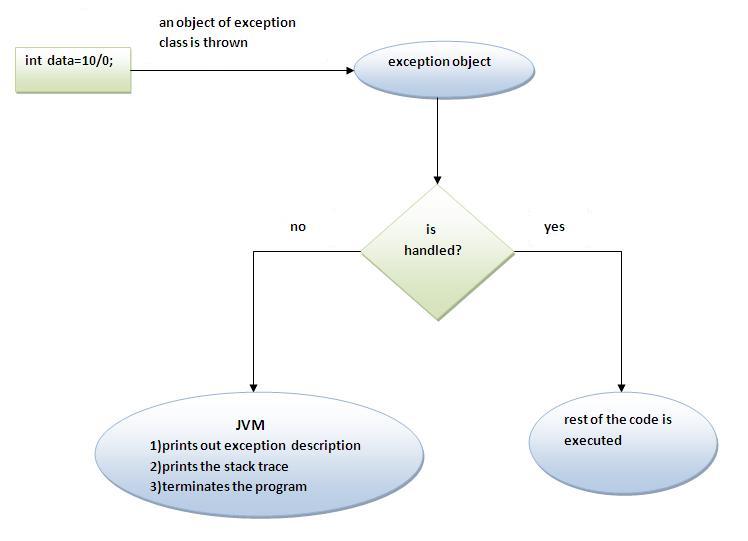
Ad by safe save. [More Info](http://www.javatpoint.com/try-catch-block) | [Hide These Ads](http://www.javatpoint.com/try-catch-block)

1. class Simple{
2. public static void main(String args[]){
3. int data=50/0;
5. System.out.println("rest of the code...");
6. }
7. }

Output:Exception in thread main java.lang.ArithmeticException:/ by zero

|  |
| --- |
| As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed. Let's see what happens behind the scene: |

### What happens behind the code int a=50/0;



|  |
| --- |
| 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. |

### Solution by exception handling

1. class Simple{
2. public static void main(String args[]){
3. try{
4. int data=50/0;
6. }catch(ArithmeticException e){System.out.println(e);}
8. System.out.println("rest of the code...");
9. }
10. }

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. |

# Multiple catch block:

|  |
| --- |
| If you have to perform different tasks at the occrence of different Exceptions, use multple catch block. |

1. <b><i>Example of multiple catch block</i></b>
3. class Excep4{
4. public static void main(String args[]){
5. try{
6. int a[]=new int[5];
7. a[5]=30/0;
8. }
9. catch(ArithmeticException e){System.out.println("task1 is completed");}
10. catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
11. catch(Exception e){System.out.println("common task completed");}
13. System.out.println("rest of the code...");
14. }
15. }

Output:task1 completed

rest of the code...

#### Rule:At a time only one Exception is occured and at a time only one catch block is executed.

#### Rule:All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

1. class Excep4{
2. public static void main(String args[]){
3. try{
4. int a[]=new int[5];
5. a[5]=30/0;
6. }
7. catch(Exception e){System.out.println("common task completed");}
8. catch(ArithmeticException e){System.out.println("task1 is completed");}
9. catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
11. System.out.println("rest of the code...");
12. }
13. }

Output:Compile-time error

# Nested try block:

|  |
| --- |
| try block within a try block is known as nested 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 |

### Syntax:

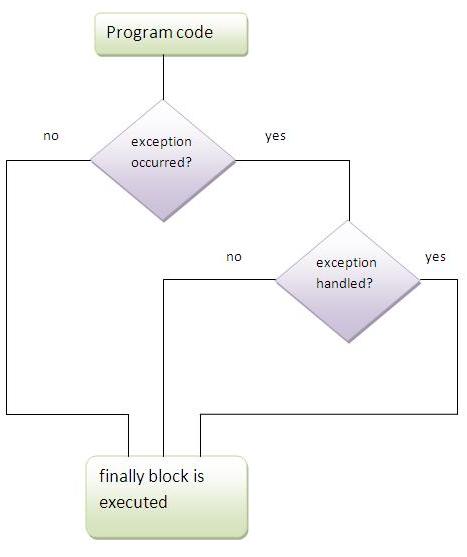
1. ....
2. try
3. {
4. statement 1;
5. statement 2;
6. try
7. {
8. statement 1;
9. statement 2;
10. }
11. catch(Exception e)
12. {
13. }
14. }
15. catch(Exception e)
16. {
17. }
18. ....

### Example:

1. <b><i>Example of nested try block</i></b>
3. class Excep6{
4. public static void main(String args[]){
5. try{
6. try{
7. System.out.println("going to divide");
8. int b =39/0;
9. }catch(ArithmeticException e){System.out.println(e);}
11. try{
12. int a[]=new int[5];
13. a[5]=4;
14. }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
16. System.out.println("other statement);
17. }catch(Exception e){System.out.println("handeled");}
19. System.out.println("normal flow..");
20. }
21. }

# finally block

|  |
| --- |
| The finally block is a block that is always executed. It is mainly used to perform some important tasks such as closing connection, stream etc. |



**Note:**Before terminating the program, JVM executes finally block(if any).

**Note:**finally must be followed by try or catch block.

### Why use finally block?

|  |
| --- |
| * finally block can be used to put "cleanup" code such as closing a file,closing connection etc. |

### case 1

***Program in case exception does not occur***

1. class Simple{
2. public static void main(String args[]){
3. try{
4. int data=25/5;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
9. finally{System.out.println("finally block is always executed");}
11. System.out.println("rest of the code...");
12. }
13. }

Output:5

finally block is always executed

rest of the code...

### case 2

***Program in case exception occured but not handled***

1. class Simple{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
9. finally{System.out.println("finally block is always executed");}
11. System.out.println("rest of the code...");
12. }
13. }

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

### case 3

***Program in case exception occured and handled***

1. class Simple{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(ArithmeticException e){System.out.println(e);}
9. finally{System.out.println("finally block is always executed");}
11. System.out.println("rest of the code...");
12. }
13. }

Output:Exception in thread main java.lang.ArithmeticException:/ by zero

finally block is always executed

rest of the code...

#### Rule: For each try block there can be zero or more catch blocks, but only one finally block.

#### Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

# throw keyword

|  |
| --- |
| The throw keyword is used to explictily throw an exception. |
| We can throw either checked or uncheked exception. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later. |

### Example of throw keyword

|  |
| --- |
| 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. |

1. class Excep13{
3. static void validate(int age){
4. if(age<18)
5. throw new ArithmeticException("not valid");
6. else
7. System.out.println("welcome to vote");
8. }
10. public static void main(String args[]){
11. validate(13);
12. System.out.println("rest of the code...");
13. }
14. }

Output:Exception in thread main java.lang.ArithmeticException:not valid

### Exception propagation:

|  |
| --- |
| An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation. |

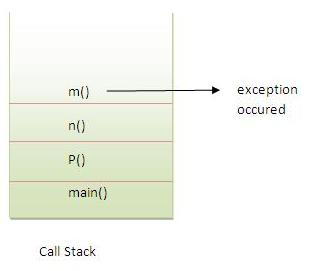
#### Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

***Program of Exception Propagation***

1. class Simple{
2. void m(){
3. int data=50/0;
4. }
5. void n(){
6. m();
7. }
8. void p(){
9. try{
10. n();
11. }catch(Exception e){System.out.println("exception handled");}
12. }
13. public static void main(String args[]){
14. Simple obj=new Simple();
15. obj.p();
16. System.out.println("normal flow...");
17. }
18. }

Output:exception handled

normal flow...



In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

#### Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

***Program which describes that checked exceptions are not propagated***

1. class Simple{
2. void m(){
3. throw new java.io.IOException("device error");//checked exception
4. }
5. void n(){
6. m();
7. }
8. void p(){
9. try{
10. n();
11. }catch(Exception e){System.out.println("exception handeled");}
12. }
13. public static void main(String args[]){
14. Simple obj=new Simple();
15. obj.p();
16. System.out.println("normal flow");
17. }
18. }

Output:Compile Time Error

# throws keyword

|  |
| --- |
| The **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. |
| Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used. |

### Syntax of throws keyword:

1. void method\_name() throws exception\_class\_name{
2. ...
3. }

### Que) Which exception should we declare?

|  |
| --- |
| **Ans)** 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. |

### Advantage of throws keyword:

Now Checked Exception can be propagated (forwarded in call stack).

***Program which describes that checked exceptions can be propagated by throws keyword.***

1. import java.io.IOException;
2. class Simple{
3. void m()throws IOException{
4. throw new IOException("device error");//checked exception
5. }
6. void n()throws IOException{
7. m();
8. }
9. void p(){
10. try{
11. n();
12. }catch(Exception e){System.out.println("exception handled");}
13. }
14. public static void main(String args[]){
15. Simple obj=new Simple();
16. obj.p();
17. System.out.println("normal flow...");
18. }
19. }

Output:exception handled

normal flow...

### Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

|  |
| --- |
| There are two cases:   1. **Case1:**You caught the exception i.e. handle the exception using try/catch. 2. **Case2:**You declare the exception i.e. specifying throws with the method. |

### Case1: You handle the exception

* In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

1. import java.io.\*;
2. class M{
3. void method()throws IOException{
4. throw new IOException("device error");
5. }
6. }

9. class Test{
10. public static void main(String args[]){
11. try{
12. Test t=new Test();
13. t.method();
14. }catch(Exception e){System.out.println("exception handled");}
16. System.out.println("normal flow...");
17. }
18. }

Output:exception handled

normal flow...

### Case2: You declare the exception

* A)In case you declare the exception, if exception does not occur, the code will be executed fine.
* B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

***A)Program if exception does not occur***

1. import java.io.\*;
2. class M{
3. void method()throws IOException{
4. System.out.println("device operation performed");
5. }
6. }

9. class Test{
10. public static void main(String args[])throws IOException{//declare exception
11. Test t=new Test();
12. t.method();
14. System.out.println("normal flow...");
15. }
16. }

Output:device operation performed

normal flow...

***B)Program if exception occurs***

1. import java.io.\*;
2. class M{
3. void method()throws IOException{
4. throw new IOException("device error");
5. }
6. }

9. class Test{
10. public static void main(String args[])throws IOException{//declare exception
11. Test t=new Test();
12. t.method();
14. System.out.println("normal flow...");
15. }
16. }

Output:Runtime Exception

### Difference between throw and throws:

|  |  |
| --- | --- |
| **throw keyword** | **throws keyword** |
| 1)throw is used to explicitly throw an exception. | throws is used to declare an exception. |
| 2)checked exception can not be propagated without throws. | 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 exception | You can declare multiple exception e.g. public void method()throws IOException,SQLException. |

### Que) Can we rethrow an exception?

Yes by throwing same exception in catch block.

# ExceptionHandling with MethodOverriding

|  |
| --- |
| There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:   * **If the superclass method does not declare an exception**   + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception. * **If the superclass method declares an exception**   + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception. |

### If the superclass method does not declare an exception

#### 1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

1. import java.io.\*;
2. class Parent{
3. void msg(){System.out.println("parent");}
4. }
6. class Child extends Parent{
7. void msg()throws IOException{
8. System.out.println("child");
9. }
10. public static void main(String args[]){
11. Parent p=new Child();
12. p.msg();
13. }
14. }

Output:Compile Time Error

#### 2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

1. import java.io.\*;
2. class Parent{
3. void msg(){System.out.println("parent");}
4. }
6. class Child extends Parent{
7. void msg()throws ArithmeticException{
8. System.out.println("child");
9. }
10. public static void main(String args[]){
11. Parent p=new Child();
12. p.msg();
13. }
14. }

Output:child

### If the superclass method declares an exception

#### 1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### Example in case subclass overridden method declares parent exception

1. import java.io.\*;
2. class Parent{
3. void msg()throws ArithmeticException{System.out.println("parent");}
4. }
6. class Child extends Parent{
7. void msg()throws Exception{System.out.println("child");}
9. public static void main(String args[]){
10. Parent p=new Child();
11. try{
12. p.msg();
13. }catch(Exception e){}
14. }
15. }

Output:Compile Time Error

### Example in case subclass overridden method declares same exception

1. import java.io.\*;
2. class Parent{
3. void msg()throws Exception{System.out.println("parent");}
4. }
6. class Child extends Parent{
7. void msg()throws Exception{System.out.println("child");}
9. public static void main(String args[]){
10. Parent p=new Child();
11. try{
12. p.msg();
13. }catch(Exception e){}
14. }
15. }

Output:child

### Example in case subclass overridden method declares subclass exception

1. import java.io.\*;
2. class Parent{
3. void msg()throws Exception{System.out.println("parent");}
4. }
6. class Child extends Parent{
7. void msg()throws ArithmeticException{System.out.println("child");}
9. public static void main(String args[]){
10. Parent p=new Child();
11. try{
12. p.msg();
13. }catch(Exception e){}
14. }
15. }

Output:child

### Example in case subclass overridden method declares no exception

1. import java.io.\*;
2. class Parent{
3. void msg()throws Exception{System.out.println("parent");}
4. }
6. class Child extends Parent{
7. void msg(){System.out.println("child");}
9. public static void main(String args[]){
10. Parent p=new Child();
11. try{
12. p.msg();
13. }catch(Exception e){}
14. }
15. }

Output:child

# Custom Exception

|  |
| --- |
| If you are creating your own Exception that is known as custom exception or user-defined exception. |

1. class InvalidAgeException extends Exception{
2. InvalidAgeException(String s){
3. super(s);
4. }
5. }
6. class Excep13{
8. static void validate(int age)throws InvalidAgeException{
9. if(age<18)
10. throw new InvalidAgeException("not valid");
11. else
12. System.out.println("welcome to vote");
13. }
15. public static void main(String args[]){
16. try{
17. validate(13);
18. }catch(Exception m){System.out.println("Exception occured: "+m);}
20. System.out.println("rest of the code...");
21. }
22. }

Output:Exception occured: InvalidAgeException:not valid

rest of the code...

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

* A user has entered 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.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

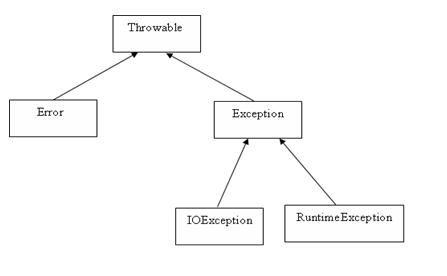
* **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
* **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
* **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.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Here is a list of most common checked and unchecked [Java's Built-in Exceptions](http://www.tutorialspoint.com/java/java_builtin_exceptions.htm).

**Exceptions Methods:**

Following is the list of important medthods available in the Throwable class.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | **public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object. |
| 3 | **public String toString()** Returns the name of the class concatenated with the result of getMessage() |
| 4 | **public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream. |
| 5 | **public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. |
| 6 | **public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace. |

**Catching Exceptions:**

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

try

{

//Protected code

}catch(ExceptionName e1)

{

//Catch block

}

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

**Example:**

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

// File Name : ExcepTest.java

import java.io.\*;

public class ExcepTest{

public static void main(String args[]){

try{

int a[] = new int[2];

System.out.println("Access element three :" + a[3]);

}catch(ArrayIndexOutOfBoundsException e){

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

}

System.out.println("Out of the block");

}

}

This would produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

Out of the block

**Multiple catch Blocks:**

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

**Example:**

Here is code segment showing how to use multiple try/catch statements.

try

{

file = new FileInputStream(fileName);

x = (byte) file.read();

}catch(IOException i)

{

i.printStackTrace();

return -1;

}catch(FileNotFoundException f) //Not valid!

{

f.printStackTrace();

return -1;

}

**The throws/throw Keywords:**

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

The following method declares that it throws a RemoteException:

import java.io.\*;

public class className

{

public void deposit(double amount) throws RemoteException

{

// Method implementation

throw new RemoteException();

}

//Remainder of class definition

}

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

import java.io.\*;

public class className

{

public void withdraw(double amount) throws RemoteException,

InsufficientFundsException

{

// Method implementation

}

//Remainder of class definition

}

**The finally Keyword**

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

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

A finally block appears at the end of the catch blocks and has the following syntax:

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}finally

{

//The finally block always executes.

}

**Example:**

public class ExcepTest{

public static void main(String args[]){

int a[] = new int[2];

try{

System.out.println("Access element three :" + a[3]);

}catch(ArrayIndexOutOfBoundsException e){

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

}

finally{

a[0] = 6;

System.out.println("First element value: " +a[0]);

System.out.println("The finally statement is executed");

}

}

}

This would produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

First element value: 6

The finally statement is executed

Note the following:

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses when ever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.

**Declaring you own Exception:**

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

* All exceptions must be a child of Throwable.
* If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
* If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

class MyException extends Exception{

}

You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

**Example:**

// File Name InsufficientFundsException.java

import java.io.\*;

public class InsufficientFundsException extends Exception

{

private double amount;

public InsufficientFundsException(double amount)

{

this.amount = amount;

}

public double getAmount()

{

return amount;

}

}

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

// File Name CheckingAccount.java

import java.io.\*;

public class CheckingAccount

{

private double balance;

private int number;

public CheckingAccount(int number)

{

this.number = number;

}

public void deposit(double amount)

{

balance += amount;

}

public void withdraw(double amount) throws

InsufficientFundsException

{

if(amount <= balance)

{

balance -= amount;

}

else

{

double needs = amount - balance;

throw new InsufficientFundsException(needs);

}

}

public double getBalance()

{

return balance;

}

public int getNumber()

{

return number;

}

}

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

// File Name BankDemo.java

public class BankDemo

{

public static void main(String [] args)

{

CheckingAccount c = new CheckingAccount(101);

System.out.println("Depositing $500...");

c.deposit(500.00);

try

{

System.out.println("\nWithdrawing $100...");

c.withdraw(100.00);

System.out.println("\nWithdrawing $600...");

c.withdraw(600.00);

}catch(InsufficientFundsException e)

{

System.out.println("Sorry, but you are short $"

+ e.getAmount());

e.printStackTrace();

}

}

}

Compile all the above three files and run BankDemo, this would produce the following result:

Depositing $500...

Withdrawing $100...

Withdrawing $600...

Sorry, but you are short $200.0

InsufficientFundsException

at CheckingAccount.withdraw(CheckingAccount.java:25)

at BankDemo.main(BankDemo.java:13)

**Common Exceptions:**

In Java, it is possible to define two catergories of Exceptions and Errors.

* **JVM Exceptions:** - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,
* **Programmatic exceptions:** - These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException.