

Object Oriented Programming in Java

Er.Sital Prasad Mandal

BCA- 2nd sem

Mechi Campus

Bhadrapur, Jhapa, Nepal

(Email : info.sitalmandal@gmail.com)

<https://ctaljava.blogspot.com/>



Text Book

1. Deitel & Dietel. -Java: How to-program-. 9th Edition. TearsorrEducation. 2011, ISBN: 9780273759168
2. Herbert Schildt. "Java: The CoriviaeReferi4.ic e 61 Seventh Edition. McGraw -Hill 2006, 1SBN; 0072263857

Handling Error / Exceptions

- 1. Basic Exceptions**
- 2. Proper use of exceptions**
- 3. User defined Exceptions**
- 4. Catching Exception: try, catch; Throwing and re-throwing: throw, throws;**
- 5. Cleaning up using the finally clause**

5. Handling Error / Exceptions



Basic Exceptions

An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

or

An Exception is an unwanted event that **interrupts the normal flow of the program**. When an exception occurs program execution gets terminated. In such cases we get a system generated error message.

The good thing about exceptions is that java developer can handle these exception in such a way so that the program doesn't get terminated suddenly and the user get a meaningful error message.

An exception can occur for many reasons. Some of them are:

- ✓ Invalid user input
- ✓ Device failure
- ✓ Loss of network connection
- ✓ Physical limitations (out of disk memory)
- ✓ Code errors
- ✓ Opening an unavailable file

5. Handling Error / Exceptions

Proper use of exceptions

There are three types of errors in java.

- 1) Syntax errors
- 2) Logical errors
- 3) Runtime errors- also called Exceptions

Syntax Errors

When compiler finds something wrong with our program, it throws a syntax error

```
int a = 9 // No semicolon, syntax errors!  
a = a + 3;  
d = 4; // Variable not declared, syntax errors
```

Logical errors

A logical error or a bug occurs when a program compiles and runs but does the wrong thing.

- Message delivered wrongly

Runtime errors

Java may sometimes encounter an error while the program is running.

These are also called Exceptions!

Ex: User supplies 'S' + 8 to a program that adds 2 numbers.

*Syntax errors and logical errors are by the **programmers**,
whereas Run-time errors are by the **users**.*

5. Handling Error / Exceptions

Proper use of exceptions

Demo for Syntax Errors, Runtime Errors & Logical Errors

```
import java.util.Scanner;
public class ErrorsDemo {
    public static void main ( String[] args ) {
        // SYNTAX ERROR DEMO
        // int a = 0 // Error: no semicolon!
        // b = 8; // Error: b not declared!

        // LOGICAL ERROR DEMO
        // Write a program to print all prime numbers between 1 to 10
        System.out.println(2);
        for (int i = 1; i < 5; i++) {
            System.out.println(2 * i + 1);
        }

        // RUNTIME ERROR
        int k;
        Scanner sc = new Scanner(System.in);
        k = sc.nextInt();
        System.out.println("Integer part of 1000 divided by k is " + 1000 / k);
    }
}
```

5. Handling Error / Exceptions

Assingment

What are the types of exceptions?

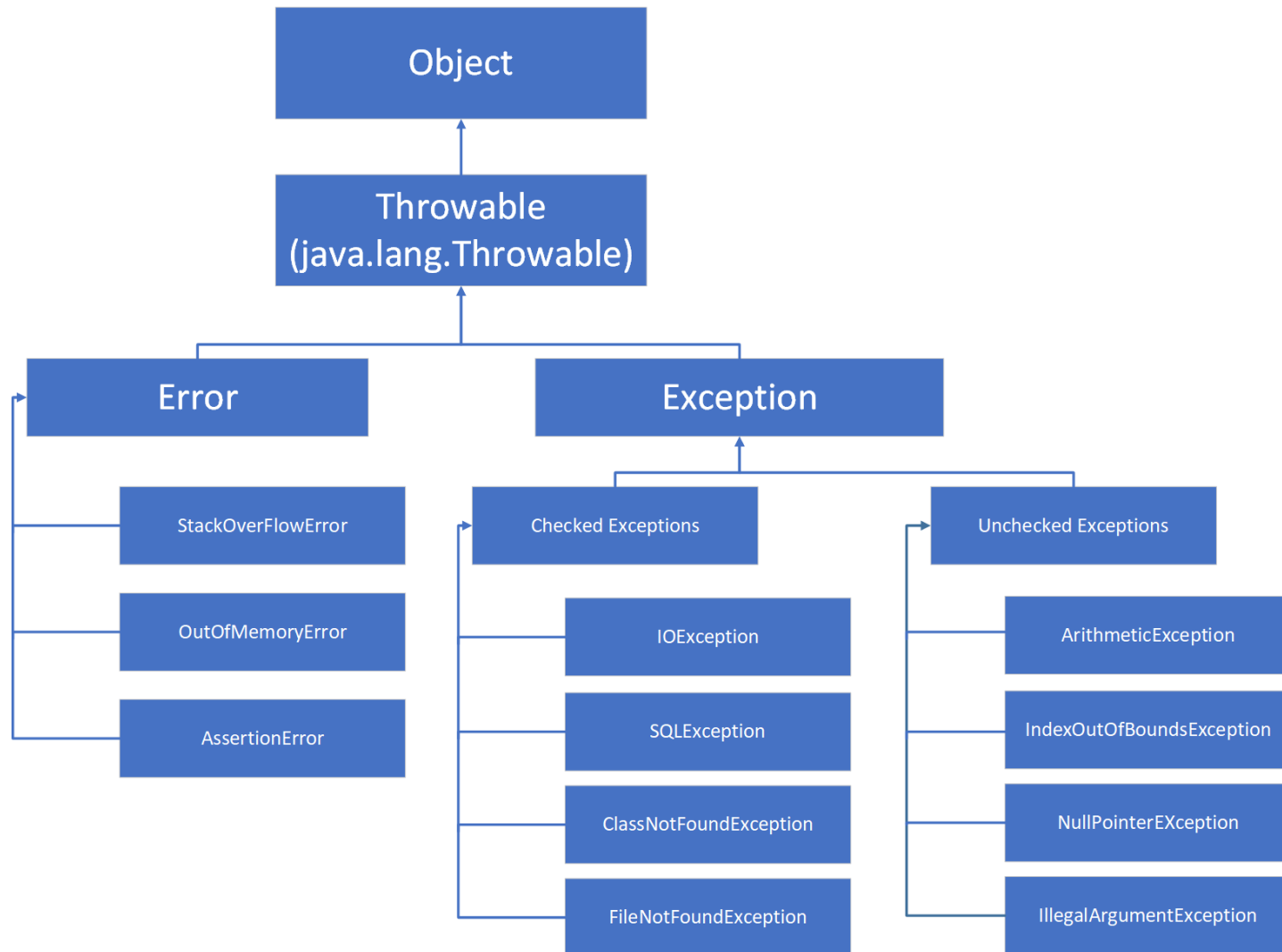
Exceptions can come in the following two exception classes:

Checked exceptions. Also called *compile-time exceptions*, the compiler checks these exceptions during the compilation process to confirm if the exception is being handled by the programmer. If not, then a compilation error displays on the system. Checked exceptions include SQLException and ClassNotFoundException, etc.

Unchecked exceptions. Also called *runtime exceptions*, these exceptions occur during program execution. These exceptions are not checked at compile time, so the programmer is responsible for handling these exceptions. Unchecked exceptions do not give compilation errors. Examples of unchecked exceptions include NullPointerException and IllegalArgumentException, etc.

5. Handling Error / Exceptions

Exception Hierarchy



5. Handling Error / Exceptions

Runtime Exceptions

1. IllegalArgumentException - Improper use of an API

```
public void IllegalArgumentException(String title) {  
    this.title = title;  
}
```

```
IllegalArgumentException(123); //errors
```

2. NullPointerException - Null pointer access

```
public class Main {  
    public static void main(String[] args) {  
        Object ref = null;  
        ref.toString(); // this will throw a NullPointerException  
    }  
}
```

5. Handling Error / Exceptions

Runtime Exceptions

3. ArrayIndexOutOfBoundsException - Out-of-bounds array access

```
class Main
{
    public static void main(String args[])
    {
        int n[]={2,4,6,8};
        System.out.println(n[5]); //error
    }
}
```

4. ArithmeticException - Dividing a number by 0 (zero)

```
public class ArithmeticException {
    public static void main(String[] args) {
        int a=0, b=4 ;
        int c = b/a;
        System.out.println("Value of c : "+c); //error
    }
}
```

5. Handling Error / Exceptions

Checked Exceptions

Checked Exceptions are checked at compile time but not at runtime.

Examples for checked exceptions are, IO Exception, SQLException, ClassNotFoundException, etc.

5. FileNotFoundException

```
import java.io.File;
import java.io.FileReader;
public class Checked
{
    public static void main (String args[])
    {
        File file = new File ("E://file.txt");
        FileReader fr = new FileReader (file); //errors
    }
}
```

5. Handling Error / Exceptions

Commonly Occurring Exceptions

The following are examples of exceptions:

1. **SQLException** is a checked exception that occurs while executing queries on a database for Structured Query Language syntax.
2. **ClassNotFoundException** is a checked exception that occurs when the required class is not found -- either due to a command-line error, a missing CLASS file or an issue with the classpath.
3. **IllegalStateException** is an unchecked exception that occurs when an environment's state does not match the operation being executed.
4. **IllegalArgumentException** : Improper use of an API
5. **NullPointerException**- Null pointer access (missing the initialization of a variable)
6. **ArrayIndexOutOfBoundsException**: Out-of-bounds array access
7. **ArithmeticException**: Dividing a number by 0
8. **IOException**: An IOException is also known as a **checked exception**. Trying to open a file that doesn't exist results in FileNotFoundException, Trying to read past the end of a file.

5. Handling Error / Exceptions



Assingment

Difference between error and exception

Errors indicate that something went wrong which is not in the scope of a programmer to handle. You cannot handle an error. Also, the error doesn't occur due to bad data entered by user rather it indicates a system failure, disk crash or resource unavailability.

Exceptions are events that occurs during runtime due to bad data entered by user or an error in programming logic. A programmer can handle such conditions and take necessary corrective actions.

Few examples:

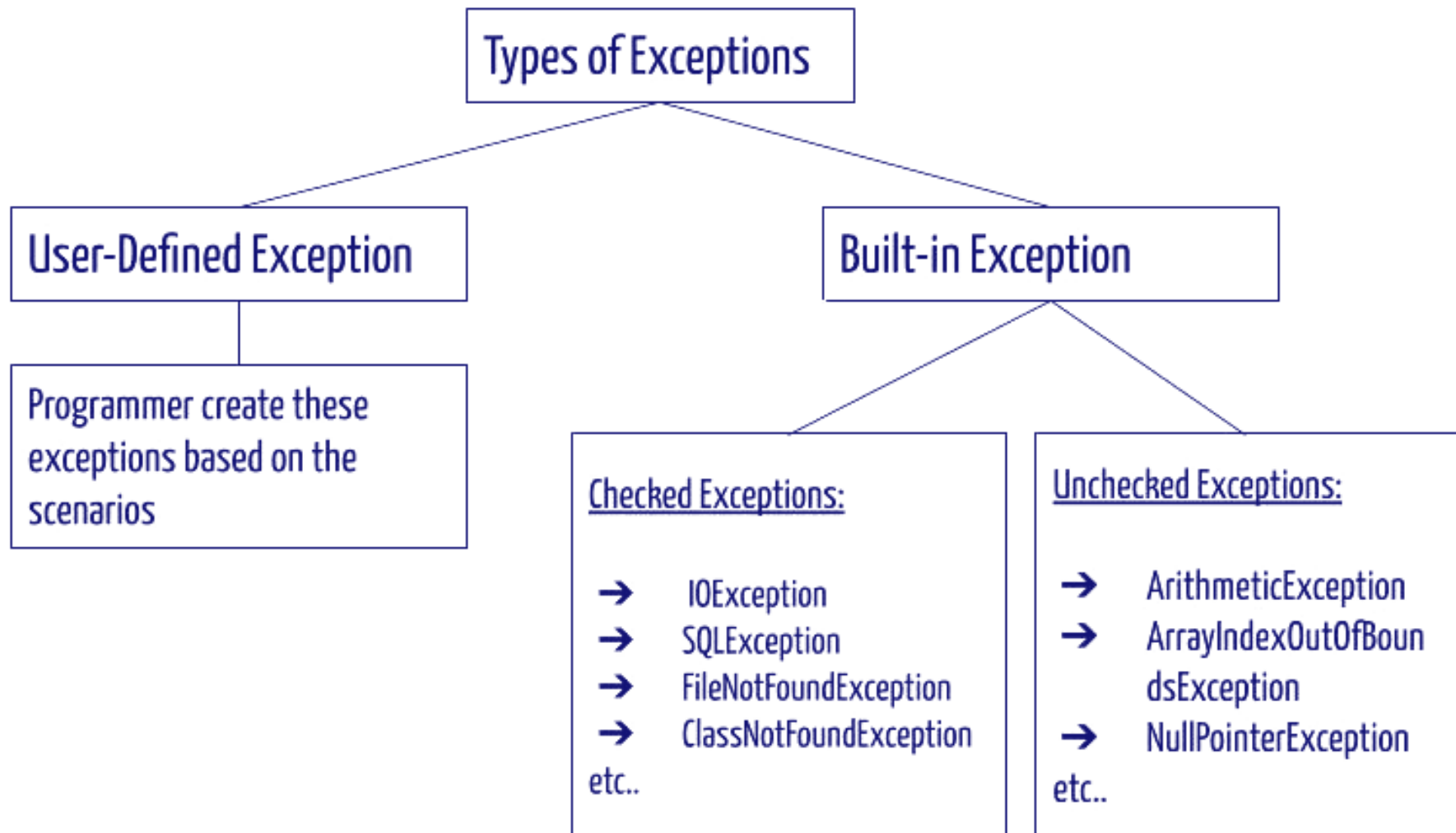
NullPointerException – When you try to use a reference that points to null.

ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

5. Handling Error / Exceptions

Assingment



5. Handling Error / Exceptions

User-Defined Exceptions

- Java user-defined exception is a custom exception created and throws that exception using a keyword '**throw**'.
- It is done by extending a class 'Exception'. An exception is a problem that arises during the execution of the program.
- In Object-Oriented Programming language, Java provides a powerful mechanism to handle such exceptions.
- *Java allows to create own exception class, which provides own exception class implementation. Such exceptions are called **user-defined exceptions** or **custom exceptions**.*


```
class UserException extends Exception {  
    int num1;  
    UserException ( int num2 ) {  
        num1 = num2;  
    }  
    public String toString () {  
        return ("Status code = " + num1);  
    }  
}
```

```
class SampleException {  
    public static void main ( String args[] ) {  
        try {  
            throw new UserException(400);  
        }  
        catch (UserException e) {  
            System.out.println(e);  
        }  
    }  
}
```

Keyword 'throw' is used to create a new Exception and throw it to catch block.

5. Handling Error / Exceptions

User-Defined Exceptions



```
// class representing custom exception
class MyCustomException extends Exception {
    /* @Override
    public String toString () {
        return "my error";
    } */
}
```

Output 1:

Caught the exception
null
rest of the code...

```
// class that uses custom exception MyCustomException
public class TestCustomException2 {
    // main method
    public static void main ( String[] args ) {
        try {
            // throw an object of user defined exception
            throw new MyCustomException();
        } catch (MyCustomException ex) {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage()); //“ex” only remove getMessage()
        }
        System.out.println("rest of the code...");
    }
}
```

Output 2:

Caught the exception
my error
rest of the code...

5. Handling Error / Exceptions

Frequently used terms in Exception handling

try: The code that can cause the exception, is placed inside try block. The try block detects whether the exception occurs or not, if exception occurs, it transfer the flow of program to the corresponding catch block or finally block. A try block is always followed by either a catch block or finally block.

catch: The catch block is where we write the logic to handle the exception, if it occurs. A catch block only executes if an exception is caught by the try block. A catch block is always accompanied by a try block.

finally: This block always executes whether an exception is occurred or not.

throw: It is used to explicitly throw an exception. It can be used to throw a checked or unchecked exception.

throws: It is used in method signature. It indicates that this method might throw one of the declared exceptions. While calling such methods, we need to handle the exceptions using try-catch block.

5. Handling Error / Exceptions

Catching Exception

The **try...catch** block in Java is used to handle exceptions and prevents the abnormal termination of the program.

Here's the syntax of a try...catch block in Java.

```
try{  
    //code  
}  
catch(exception) {  
    // code  
}
```

The **try block** includes the code that might generate an exception.

The **catch block** includes the code that is executed when there occurs an exception inside the try block.

<https://beginnersbook.com/2013/04/nested-try-catch/>

5. Handling Error / Exceptions



Catching Exception try catch

Example: Java try...catch block

```
class TryCatchDemo {  
    public static void main(String[] args) {  
  
        try {  
            int divideByZero = 5 / 0;  
            System.out.println("Print" + divideByZero );  
        }  
  
        catch (ArithmeticException e) {  
            System.out.println("ArithmeticException => " + e.getMessage());  
        }  
    }  
}
```

Output

ArithmeticException => / by zero

In the above example, notice the line,

int divideByZero = 5 / 0;

5. Handling Error / Exceptions



Catching Exception Finally block

Can we have a try block without a catch block in Java?

Yes, It is possible to have a try block without a catch block by using a **final block**.

As we know, a **final** block will always **execute even there is an exception** occurred in a try block, except `System.exit()` it will execute always.

Syntax of try block with finally block

```
try{  
    //statements that may cause an exception  
}
```

```
finally{  
    // System.out.println("Finally block is always executed");  
    //statements that execute whether the exception occurs or not  
}
```

5. Handling Error / Exceptions

Catching Exception try-catch-finally in Java

```
try{  
    //statements that may cause an exception  
}  
catch(Exception e){  
    //statements that will execute if exception occurs  
}  
finally{  
    //statements that execute whether the exception occurs or not  
}
```

5. Handling Error / Exceptions

Catching Exception try-catch-finally in Java

```
public class TryWithFinally {  
    public static int method() {  
        try {  
            System.out.println("Try Block with return type");  
            return 10;  
        } finally {  
            System.out.println("Finally Block always execute");  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println(method());  
    }  
}
```

Output:

```
Try Block with return type  
Finally Block always execute  
10
```

5. Handling Error / Exceptions

Catching Exception Multiple catch blocks in Java

```
try{
    ....
} catch(ArithmeticException e){
    ....
} catch(Exception e) {
    //this is generic exception handler
    //it can handle any exception
}
```

If any exception occurs other than ArithmeticException

If ArithmeticException occurs in try block

The diagram illustrates the execution flow of a try-catch block. A dashed arrow from the text 'If any exception occurs other than ArithmeticException' points to the second catch block, `} catch(Exception e) {`. Another dashed arrow from the text 'If ArithmeticException occurs in try block' points to the first catch block, `} catch(ArithmeticException e){`. The code block shows a try block followed by two catch blocks: one for `ArithmeticException` and a generic one for `Exception`. The generic catch block contains comments: `//this is generic exception handler` and `//it can handle any exception`.

5. Handling Error / Exceptions



Catching Exception

Throwing and re-throwing: throw,
throws;

5. Handling Error / Exceptions

Catching Exception

throw keyword

It is used to throw the user defined or customized exception object To the JVM explicitly.

```
public class ThrowDemo1 {  
    public static void main ( String[] args ) {  
        //System.out.println(10/0);  
        // mainly used for userdefine Exception  
        throw new ArithmeticException(" divided by zero");  
    }  
}
```

*Exception in thread "main" java.lang.ArithmeticException: divided by zero
at Unit4.ThrowDemo1.main(ThrowDemo1.java:7)*

5. Handling Error / Exceptions

Catching Exception

throws keyword

It is used when we doesn't want to handle the exception and try to send the exception to The JVM or other method.

```
public class ThrowsDemo1 {  
    public static void main ( String[] args ) throws InterruptedException  
{  
        for (int i = 1; i<=10; i++){  
            System.out.println(i);  
            Thread.sleep(1000); // 1 sec delay  
        }  
    }  
}
```

Output:

1
2
3
4
5

```
for (int i = 1; i<=5; i++){  
    //normally terminate  
    try {  
        System.out.println(i);  
        Thread.sleep(1000); // 1 sec delay  
    }catch (InterruptedException e){  
        System.out.println(e);  
    }  
}
```

5. Handling Error / Exceptions

Catching Exception

Re-throwing Exception

```
public class RethrowingException {  
    public static void main ( String[] args ) {  
        try{  
            System.out.println(10/0);  
        } catch (ArithmeticException a){  
            //System.out.println(a);  
            throw new NullPointerException();  
        }  
    }  
}
```

Output:

Exception in thread "main" java.lang.NullPointerException

5. Handling Error / Exceptions

Catching Exception

Throwing and re-throwing

```
public class throwthrowsDemo {
    void div(int a, int b) throws ArithmeticException{
        if (b==0){
            throw new ArithmeticException();
        }
        else {
            int c = a/b;
            System.out.println(c);
        }
    }
    public static void main ( String[] args ) throws ArithmeticException {
        throwthrowsDemo t = new throwthrowsDemo();
        try {
            t.div(20,0);
        }
        catch (Exception e){
            System.out.println("b is zero");
        }
    }
}
```

5. Handling Error / Exceptions

Assignment

Q. Difference between throw & throws ?

Ans →

throw

- ① throw keyword is used to throw an exception object explicitly.

void m1() {
ex → throw new AEC();
}

- ② throw keyword always present inside method body.

- ③ We can throw only one exception at a time.

throw new AEC()
object (instance)

- ④ throw is followed by an instance.

throws

- ① throws keyword is used to declare an exception as well as by pass the caller.

void m1() throws AEC;
? caller signature
}

- ② ~~throws~~ keyword ~~is~~ always used with method Signature.

- ③ We can handle multiple exceptions using throws keyword.

m1() throws AEC, NPE, etc;
{
}

- ④ throws is followed by class.

5. Handling Error / Exceptions

Assignment

throw	throws
1. Java throw keyword is used to explicitly throw an exception	1. Java throws keyword is used to declare an exception.
2. <pre>void m(){ throw new ArithmeticException("sorry"); }</pre>	2. <pre>void m()throws ArithmeticException{ //method code }</pre>
3. Checked exception cannot be propagated using throw only.	3. Checked exception can be propagated with throws.
4. Throw is followed by an instance.	4. Throw is followed by a class.
5. Throw is used within the method.	5. Throws is used with the method signature.
6. You cannot throw multiple exceptions.	6. You can declare multiple exceptions e.g. <pre>public void method()throws IOException,SQLException.</pre>

5. Handling Error / Exceptions



Cleaning up using the finally clause

A **finally block** contains all the crucial(significant) statements that must be executed whether exception occurs or not.

The statements present in this block will always execute whether exception occurs in try block or not such as closing a connection, stream etc.

```
try {  
    //Statements that may cause an exception  
}  
catch {  
    //Handling exception  
}  
finally {  
    //Statements to be always executed  
}
```

5. Handling Error / Exceptions

Cleaning up using the finally clause

```
class Main {  
    public static void main(String[] args) {  
        try {  
            System.out.println("Try Block");  
        }  
  
        finally {  
            System.out.println("Finally block is always executed");  
        }  
    }  
}
```

Output

Try Block Finally Block

Finally block is always executed

5. Handling Error / Exceptions



Cleaning up using the finally clause

```
class JavaFinally
{
    public static int myMethod()
    {
        try {
            return 112;
        }
        finally {
            System.out.println("This is Finally block");
            System.out.println("Finally block ran even after return statement");
        }
    }

    public static void main(String args[])
    {
        System.out.println(JavaFinally.myMethod());
    }
}
```

Another example of finally block and return statement

Output

```
This is Finally block
Finally block ran even after return statement
112
```

5. Handling Error / Exceptions



Assignment

Cases when the finally block doesn't execute

The circumstances that prevent execution of the code in a finally block are:

- The death of a Thread
- Using of the `System. exit()` method.
- Due to an exception arising in the finally block.

5. Handling Error / Exceptions

Assignment

Example 3: When exception occurs in try block and handled properly in catch block.

```
public class FinallyExample {  
    public static void main(String args[]){  
        try{  
            System.out.println("First statement of try block");  
            int num=45/0;  
            System.out.println(num);  
        }  
        catch(ArithmeticException e){  
            System.out.println("ArithmeticException");  
        }  
        finally{  
            System.out.println("finally block");  
        }  
        System.out.println("Out of try-catch-finally block");  
    }  
}
```

Output:

```
First statement of try block  
ArithmeticException  
finally block  
Out of try-catch-finally block
```

5. Handling Error / Exceptions



Later on..

Finally and Close()

```
...
InputStream input = null;
try {
    input = new FileInputStream("inputfile.txt");
}
finally {
    if (input != null) {
        try {
            in.close();
        } catch (IOException exp) {
            System.out.println(exp);
        }
    }
}
...
```

```
....
try{
    OutputStream osf = new FileOutputStream( "filename" );
    OutputStream osb = new BufferedOutputStream(opf);
    ObjectOutputStream op = new ObjectOutputStream(osb);
    try{
        output.writeObject(writableObject);
    }
    finally{
        op.close();
    }
}
catch(IOException e1){
    System.out.println(e1);
}
...
```