Exception Handling:

1. Introduction
2. Runtime stack mechanism
3. Default Exception Handling in Java
4. Exception Hierarchy
5. Customized exception handling by using try catch
6. Control flow in try catch
7. Method to print exception information
8. try with multiple catch blocks
9. finally block
10. difference between final, finally, finalize
11. control flow in try-catch-finally
12. control flow in nested try-catch-finally
13. various possible combinations of try catch finally
14. throw keyword
15. throws keyword
16. Exception handling keywords summary
17. Various possible compile time errors in exception handling
18. Customized or user defined exceptions
19. Top -10 exceptions
20. 1.7 version enhancements
    1- try with resources
    2- multi-catch block

1. Introduction

An unexpected, unwanted event that disturbs the normal flow of the program is called exception.
Ex:
TirePunchuredException
SleepingException
FileNotFoundException
Etc.

Question: What is the main purpose of Exceptional Handling?
Answer:

1- Graceful Termination meaning of the program.
2- It is highly recommended to handle exceptions and the main objective of exception is graceful termination of the program.

Question: What is the meaning of Exception Handling?
Answer: Defining alternative way to continue rest of the program normally i.e. called exception handling.
Example: If something goes wrong then having alternative way to continue our program normally i.e. the concept of exceptional handling.

```
try {
        Read data from remote file locations at London.
}catch(FileNotFoundException e){
        Use local file & continue rest of the program normally.
}
```

Exceptional handling doesn’t' mean repair an exception, we have to provide alternative way to continue rest of the normally, i.e. the concept of exceptional handling.

For example: Our programming requirement is to read the data from remote file location at London at runtime if London file is not available our program should not be terminated abnormally. Then we have to provide some local file to continue rest of the program normally. This way of define alternative is nothing but exceptional handling.

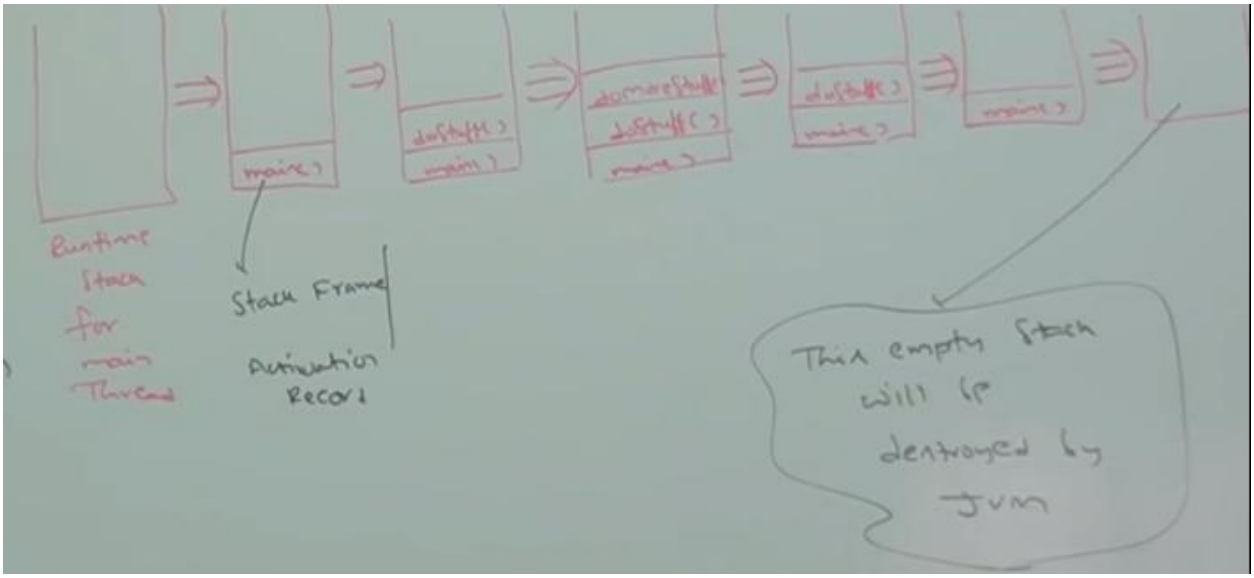
2. Runtime stack mechanism

Example:

```
package westpac.com.au;

public class RunTime_Stack_Maechanism_in_Exceptional_Handling {
	public static void main(String[] args) {
		doStuff();
	}
	public static void doStuff(){
		doMoreStuff();
	}
	public static void doMoreStuff(){
		System.out.println("Hello Ankur...");
	}
}
```

Question: In the above program how many thread is there?
Answer : Only one thread i.e. main thread. In the above program JVM created with one Runtime Stack. And it is the Runtime stack for main Thread.



For every thread JVM will create a runtime stack. Each and every method performed that thread will be stored in the corresponding stack.
Each entry stack is called stack frame/ activation record. After completing every method called the corresponding entry form stack will be removed.

After completing all method called the stack will become empty and that empty stack will be destroy by JVM just before termination the thread.

3. Default Exception Handling in Java

1. Inside a method, if any exception occurs the method in which is raised is responsible to create exception object by including the following information.
    1. Name of Exception
    2. Description of Exception
    3. Location at which exception occurs (stack trace)

2. After creating exception object method handover that object to the JVM.

3. JVM will check whether the method contains any exception handling code or not. If the method doesn't exception handling code then JVM terminate that method abnormally and the remove corresponding from the stack.

4. Then JVM identifies caller method and checks whether caller method contains any handling code or not.

5. If the caller method doesn't contains handling code then JVM terminate the caller method also abnormally and remove the corresponding entry from the stack.

6. This process will be continued until main method and if the main method also doesn't contain handling code then JVM terminates main method also abnormally and remove corresponding entry form stack.

7. Then JVM handovers responsibility of exception handling to default Exception Handler, which is the part of JVM.

8. Default exception handler print exception information in the following format and terminates program abnormally.

Exception in Thread "XXX" name of Exception: Description Stack Trace.

Example 1:

```
package westpac.com.au_Exception_Dafault;

public class Default_Exception {
	public static void main(String[] args) {
		doStuff();
	}
	public static void doStuff(){
		doMoreStuff();
	}
	public static void doMoreStuff(){
		System.out.println("Hello Ankur...");
		System.out.println(10/0);
	}
}
```

OutPut:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at westpac.com.au_Exception_Dafault.Default_Exception.doMoreStuff(Default_Exception.java:12)
at westpac.com.au_Exception_Dafault.Default_Exception.doStuff(Default_Exception.java:8)
at westpac.com.au_Exception_Dafault.Default_Exception.main(Default_Exception.java:5)
```

Example 2:

```
package westpac.com.au_Exception_Dafault;

public class Default_Exception {
	public static void main(String[] args) {
		doStuff();
	}
	public static void doStuff(){
		doMoreStuff();
		System.out.println(10/0);
	}
	public static void doMoreStuff(){
		System.out.println("Hello Ankur...");
	}
}
```

Output:

```
Hello Ankur...
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at westpac.com.au_Exception_Dafault.Default_Exception.doStuff(Default_Exception.java:9)
	at westpac.com.au_Exception_Dafault.Default_Exception.main(Default_Exception.java:5)
```

Example 3:

```
public class Default_Exception {
	public static void main(String[] args) {
		doStuff();
		System.out.println(10/0);
	}
	public static void doStuff(){
		doMoreStuff();
		System.out.println("Hello Ankit Babu...");
	}
	public static void doMoreStuff(){
		System.out.println("Hello Ankur...");
	}
}
```

Output:

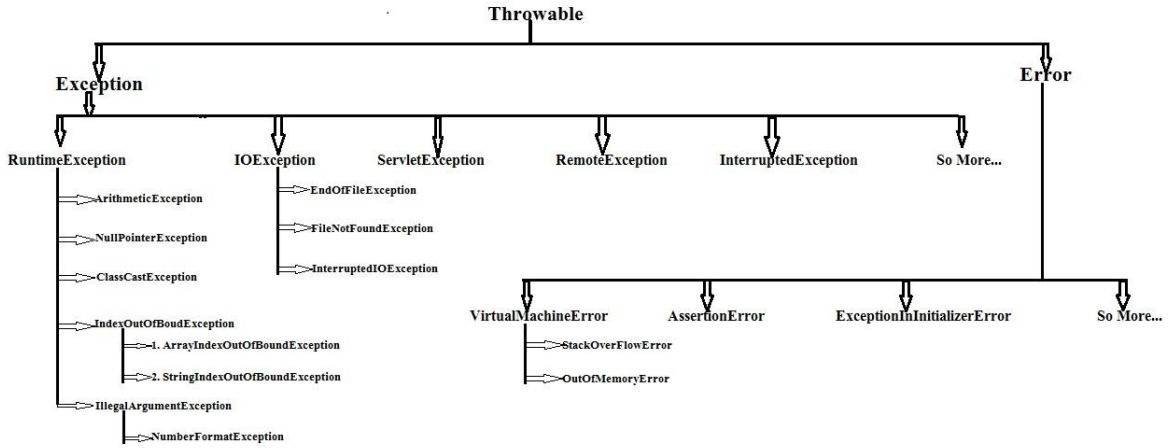
```
Hello Ankur...
Hello Ankit Babu...
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at westpac.com.au_Exception_Dafault.Default_Exception.main(Default_Exception.java:6)
```

Note:

1. In a program, at least one method terminate abnormally then the program termination is abnormal termination.
2. If all method terminated normally then only program termination is normal termination.

4. Exception Hierarchy



Throwable class access root for Java exception hierarchy. Throwable class define 2 child classes:

1. Exception
2. Error

1. Exception:

Most of the time exception are caused by our program. And these are recoverable. For example: If our programming requirement is to read data from remote file locating at London at runtime if Remote file is not available then we will get runtime exception saying file not found exception. If FileNotFound exception occurs then we can provide local file and continue rest of the program normally.

```
try {
        Read data from Remote file locating file at London.
}catch(FileNotFoundException e){
        Use Local file and continue rest of the program normally.
}
```

2. Error:

Most of the time error are not caused by our program. And these are due to lack of system resources. Errors are non-recoverable.
For Example: If OutOfMemoryError occurs, being a programmer we can't do anything and program will be terminated abnormally.
System Admin/Server Admin is responsible to increase heap memory.

1. Exception Types:

1. RuntimeException:
   1) ArithmeticException
   2) NullPointerException
   3) ClassCastException
   4) IndexOutOfBoudException
      1. ArrayIndexOutOfBoundException
      2. StringIndexOutOfBoundException
   5) IllegalArgumentException
      1. NumberFormatException

2. IOException
   1 EndOfFileException
   2 FileNotFoundException
   3 InterruptedIOException

3. ServletException
4. RemoteException
5. InterruptedException
6. So More...

2. Error Types:

1. VirtualMachineError
   1. StackOverFlowError
   2. OutOfMemoryError
2. AssertionError
3. ExceptionInInitializerError
4. So More...

Example:

```
package westpac.com.au_Exception_Print_Writer_with_File_not_Found;

import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class File_Not_Found_Exception {
	public static void main(String[] args) {
		PrintWriter printWriter;
		try {
			printWriter = new PrintWriter("Ankur.txt");
			printWriter.println("Hello Ankur...");

		} catch (FileNotFoundException e) {
			e.printStackTrace();
		}
	}
}
```

//
//

Checked Exception vs. Unchecked Exception:

The Exception which are checked by compiler for smooth execution of the program are called Checked Exception.

Ex:
HallTicketMissingException
PenNotWorkingException
FileNotFoundException etc.

In our program if there is a chance of rising checked exception then compulsory we should handle that checked exception(either by try catch or by throws keyword) otherwise we'll get compile time error.

The exceptions which are not checked by compiler whether programmer handling or not. Such type of exception are called unchecked exception.
Arithmetic Exception
BombBlastException etc.

Note:
1. Whether it is checked or unchecked every exception occurs yet runtime only. There is no chance of occurring any exception yet compile time.

2. Runtime exception and its child classes, Errors and its child classes are unchecked except these remaining are checked.

Fully Checked vs. Partially Checked:

1. A checked exception is said to be fully checked if and only if all its child classes also checked.
Example: IOException, IntterruptedException

2. A checked exception is said to be partially checked if and only if some its child classes are unchecked.
Example: Exception, Throwable.

Note:
The only possible partially checked exception in java are:
1. Exception
2. Throwable

Describe the behavior of following Exception:
IOException ➔Checked (Fully)
RuntimeException ➔UnChecked
InterruptedException ➔Checked (Fully)
Error ➔Unchecked
Throwable ➔Checked (Partially)
ArithmeticException ➔Unchecked
NullPointerException ➔Unchecked
Exception ➔Checked (Partially)
FileNotFoundException ➔Checked (Fully)

5. Customized exception handling by using try catch

1. It is highly recommended to handle exceptions.
2. The code which may raise an exception risky code and we have to define that code inside try block. And corresponding handling code, we have to define inside catch block.

Syntax:

```
try{
```

```
	// Risky Code
}catch(Exception e){
	// Handling Code
}
```

Without Try-Catch Block Example:

```
package westpac.com.au_Exception_Without_try_catch;

import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class Without_Try_Catch {
	public static void main(String[] args) {
		System.out.println("Statement 1");
		System.out.println(10/0);
		System.out.println("Statement 3");
	}

}
```

Outut:

```
Statement 1
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at westpac.com.au_Exception_Without_try_catch.Without_Try_Catch.main(Without_Try_Catch.java:9)
```

With Try-Catch Block Example:

```
package westpac.com.au_Exception_With_try_catch;

public class With_Try_Catch {
	public static void main(String[] args) {
		System.out.println("Statement 1");
		try{
			System.out.println(10/0);
		}catch(ArithmeticException e){
			System.out.println(10/2);
		}
		System.out.println("Statement 3");
	}
}
```

Output:

```
Statement 1
5
Statement 3
```

6. Control flow in try catch

Syntax:

```
	try{
		// Statement 1
		// Statement 2
```

```
        // Statement 3
}catch(Exception e){
        // Statement 4
}
// Statement 5
```

Case 1: If there is no exception

1,2,3,5 ➔Normal Termination (N T)

Case 2: If exception raised at **Statement 2** and corresponding catch block matched then the statement flow is:

```
try{
        // Statement 1
        // Statement 2
        // Statement 3
}catch(Exception e){
        // Statement 4
}
// Statement 5
```

1, 4, 5 ➔NT

Case 3: If exception raised at **Statement 2** and corresponding catch block is not matched then the statement flow is:

```
try{
        // Statement 1
        // Statement 2
        // Statement 3
}catch(Exception e){
        // Statement 4
}
// Statement 5
```

1, Abnormal Termination (AT)

Case 3: If exception raised at **Statement 4 or Statement 5** then it is always Abnormal Termination:

```
try{
        // Statement 1
        // Statement 2
        // Statement 3
}catch(Exception e){
        // Statement 4
}
// Statement 5
```

Note:

1. Within the try block if anywhere exception raised then rest of the try block would not be executed even though we handled the that exception. Hence within the try block we have to take only risky code and length of try block should be as less as possible.
2. In addition to try block there may be chance to rising an exception inside catch and finally blocks.
3. If any statement which it is not part of try block and raised an exception then it is always abnormal terminations

7. Method to print Exception Information

Throwable class define the following methods to print exception information.

Method	Pintable Format
1. printStaceTrace()	Name of Exception: Description : Stace Trace
2. toString()	Name of Exception: Description
3. getMessage()	Description

Note: Internally default exception handler will use printStaceTrace method to print exception information method to the console.

Example:

```
package westpac.com.au_Exception_10_print_Exception_Information;

public class Print_Exception_Information {
	public static void main(String[] args) {
		System.out.println("Statement 1");
		try{
			System.out.println(10/0);
		}catch(ArithmeticException e){
			e.printStackTrace();
			System.out.println(e);
			System.out.println(e.toString());
			System.out.println(e.getMessage());

		}
		System.out.println("Statement 3");
	}
}
```

Output:

```
Statement 1
java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
/ by zero
Statement 3
	at
westpac.com.au_Exception_10_print_Exception_Information.Print_Exception_Information.main(Print_Exception_Information.java
:7)
```

8. try with multiple catch blocks

The way of handling an exception is varied from exception to exception. Hence for every exception type it is highly recommended to take separate catch block i.e. try with multiple catch block is always possible and recommended to use.

```
try{
	// Risky Code
}catch (Exception e) {
	// Default Exception-Handling
}
```

Worst Programming Practice

```
try{
	// Risky Code
	System.out.println(10/0);
}catch(ArithmeticException e){
	// Perform Alternative arithmetic operations
}
catch (SQLException e) {
	// Use MySQL DB instead of Oracle.
}
catch (FileNotFoundException e) {
	// Use Local File instead of Remote File.
}
catch (Exception e) {
	// Default Exception-Handling
}
```

Best Programming Practice

If try with multiple catch block present then the order of catch block is very important. We have to take child first and then parent otherwise we will get compile time error saying

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Unreachable catch block for ArithmeticException. It is already handled by the catch block for Exception
```

Example:

```
import java.io.FileNotFoundException;
import java.sql.SQLException;

public class Muliple_Try_Catch {
	public static void main(String[] args) {
		try{
			// Risky Code
			System.out.println(10/0);
		}catch (Exception e) {
			// Default Exception-Handling
			System.out.println("Default Exception-Handling Operation");
		}
		catch(ArithmeticException e){
			// Perform Alternative arithmetic operations
			System.out.println("ArithmeticException Operation");
```

```
		}
	}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Unreachable catch block for ArithmeticException. It is already handled by the catch block for Exception

	at westpac.com.au_Exception_11_Multiple_try_catch.Muliple_Try_Catch.main(Muliple_Try_Catch.java:14)
```

Example:

```
import java.io.FileNotFoundException;
import java.sql.SQLException;

public class Muliple_Try_Catch {
	public static void main(String[] args) {
		try{
			// Risky Code
			System.out.println(10/0);
		}catch(ArithmeticException e){
			// Perform Alternative arithmetic operations
			System.out.println("ArithmeticException Operation");
		}
		catch (Exception e) {
			// Default Exception-Handling
			System.out.println("Default Exception-Handling Operation");
		}
	}
}
```

Output:

```
ArithmeticException Operation
```

We can't 2 catch block for the same exception otherwise we'll get compile time error.

Example:

```
import java.io.FileNotFoundException;
import java.sql.SQLException;

public class Muliple_Try_Catch {
	public static void main(String[] args) {
		try{
			// Risky Code
			System.out.println(10/0);
		}
		catch(ArithmeticException e){
			// Perform Alternative arithmetic operations
			System.out.println("ArithmeticException Operation");
		}catch(ArithmeticException e){
```

```
                        // Perform Alternative arithmetic operations
                        System.out.println("ArithmeticException Operation");
                }
        }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Unreachable catch block for ArithmeticException. It is already handled by the catch block for
ArithmeticException

        at
westpac.com.au_Exception_11_Multiple_try_catch.Muliple_Try_Catch.main(Muliple_Try_Catch.java:18)
```

9. Difference between final, finally, finalize

final Modifier:

final is a modifier applicable for classes, methods and variables. If a class a declared as a final then we can't extends that class i.e. we can't create child class for that class i.e. inheritance is not possible for final classes.

If a method is final then we can't override that method in child class.

If a variable declared as final then we can't perform re-assignment for that variable.

finally Block:

finally is a block always associated with try catch to maintain cleanup code.

```
try{
        // Risky Code
}catch (Exception e) {
        // Default Exception-Handling
}finally{
        // Cleanup Code
}
```

The specialties of finally block is it will be executed always irrespective of whether exception is raised or not raised and whether handle or not handle.

finalize() Method:

finalize is a method always invoked by garbage collector just before destroying an object to perform cleanup activities

Once finalize method complete immediately garbage collector destroy that object.

Note:

finally block is responsible to perform cleanup activities related to try block i.e. whatever resources we open as a part of try block will be closed inside the finally block.

Whereas finalize method is responsible to perform cleanup activities related to object i.e. whatever resources associated with object will be de-allocated before destroying an object by using finalize method.

10. Various possible combinations of try catch finally

1. In try catch finally block order is important.
2. Whenever we are writing try compulsory we should write either catch or finally otherwise we'll get compile time Error i.e. try without catch or finally is invalid.
3. Whenever we are writing catch block compulsory try block must be required i.e. catch without try is invalid.
4. Whenever we are writing finally block compulsory we should write try block i.e. finally without try is invalid.
5. Inside try catch finally blocks we can declare try-catch and finally blocks i.e. nesting of try-catch-finally is allowed.
6. For try catch and finally block curly braces are mandatory.

Case 1:

```
try{

}catch(x e){

}
```

Case 2:

```
try{

}catch(x e){

}catch(y e){

}
```

Case 3:

```
try{

}catch(x e){

}catch(y e){

}
```

CE: exception x has already been caught

Case 4:

```
try{

}catch(x e){

}finally{

}
```

Case 5:

```
try{

}finally{

}
```

Case 6:

```
try{

}catch(x e){

}
try{

}catch(y e){

}
```

Case 7:

```
try{

}catch(x e){

}
try{

}finally{

}
```

Case 8:

```
try{

}
```

CE: try without catch (or) finally.

Case 9:

```
catch(x e){

}
```

CE: catch without try.

Case 10:

```
finally{

}
```

CE: finally without try.

Case 11:

```
try{

}finally{

}
catch(x e){

}
```

CE: catch without try.

Case 12:

```
try{

}
System.out.println("Hello...");
catch(x e){

}
```

CE1: try without catch (or) finally.
CE2: catch without try.

Case 13:

```
try{

}
catch(x e){

}
System.out.println("Hello...");
catch (y e) {

}
```

CE: catch without try.

Case 14:

```
try{

}
catch(x e){

}
System.out.println("Hello...");
finally{

}
```

CE: finally without try.

Case 15:

```
try{
	try{

	}catch(x e){

	}
}catch(x e){

}
```

Case 16:

```
try{
	try{

	}
}
catch(x e){

}
```

CE: try without catch (or) finally.

Case 17:

```
try{
	try{

	}finally{

	}
}catch(x e){

}
```

Case 18:

```
try{

}catch(x e){
	try{

	}finally{

	}
}
```

Case 19:

```
try{

}catch(x e){
	finally{

	}
}
```

CE: finally without try.

Case 20:

```
try{

}
catch(x e){

}finally{
	try{

	}
	catch(x e){

	}
}
```

Case 21:

```
try{

}
catch(x e){

}finally{
	finally{

	}
```

```
}
```

CE: finally without try.

Case 22:

```
try{

}
catch(x e){

}finally{

}finally{

}
```

CE: finally without try.

Case 23:

```
try
	System.out.println("try");

catch(x e){
	System.out.println("catch");
}finally{

}
```

Note: Curly Braces are Mandatory in try.

Case 24:

```
try{

}
catch(x e)
	System.out.println("catch");
finally{

}
```

Note: Curly Braces are Mandatory in catch.

Case 25:

```
try{
```

```
}
catch(x e){

}finally
            System.out.println("finally");
```

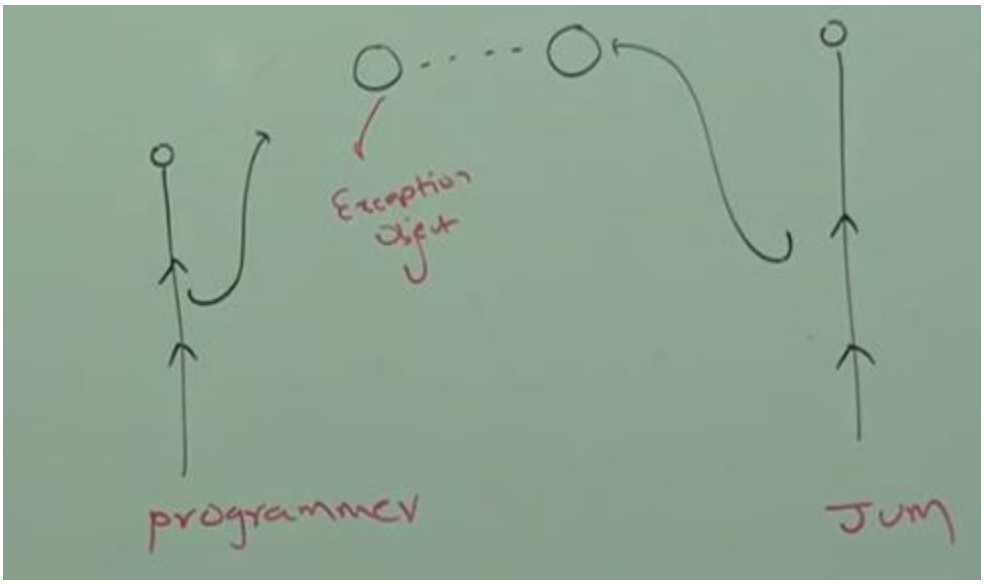
✘

Note: Curly Braces are Mandatory.

11. throw keyword

Sometimes we can create exception object explicitly we can handover to the JVM manually. For this we have to use throw keyword.
throw new ArithmeticException("/ by zero);

Creation of ArithmeticException object explicitly.
Hand-over our created exception object to the JVM manually. For this purpose we use throw keyword.
Hence the main objective of throw keyword is to Hand-over our created exception object to the JVM manually.



Hence the result of following two program are exactly same.

Example 1:

```
package westpac.com.au_Exception_11_Multiple_try_catch;

public class Throw_Keyword {
	public static void main(String[] args) {
		System.out.println(10/0);

	}
```

```
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at
westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:5)
```

In this case main method is responsible to create exception object and hand-over to the JVM.

Example 2:

```
package westpac.com.au_Exception_11_Multiple_try_catch;

public class Throw_Keyword {
	public static void main(String[] args) {
		throw new ArithmeticException("/ by Zero");
	}
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by Zero
	at
westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:5)
```

In this case programmer is creating exception object explicitly and hand-over to the JVM manually.

Example 3:

```
package westpac.com.au_Exception_11_Multiple_try_catch;

public class Throw_Keyword {
	public static void main(String[] args) {
		throw new ArithmeticException("/ by Zero Expicitly...");

	}
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by Zero Expicitly...
	at
westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:5)
```

Example :

```
package westpac.com.au_Exception_11_Multiple_try_catch;

public class Throw_Keyword {
	static ArithmeticException e=new ArithmeticException();
	public static void main(String[] args) {
		throw e;
	}
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException
	at westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.<clinit>(Throw_Keyword.java:4)
```

Case 1:

```
package westpac.com.au_Exception_11_Multiple_try_catch;

public class Throw_Keyword {
	static ArithmeticException e;
	public static void main(String[] args) {
		throw e;
	}
}
```

Note: `throw e;` if e refers null then we'll get **NullPointerException**.

Output:

```
Exception in thread "main" java.lang.NullPointerException
	at westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:6)
```

Case 2:

After throw statement we are not allowed to write any statement directly otherwise we'll get compile time error saying Unreachable statement.

Example:

```
public class Throw_Keyword {
	public static void main(String[] args) {
		System.out.println(10/0);
		System.out.println("Hello...");
	}
}
```

Output:

```
RE: Exception in thread "main" java.lang.ArithmeticException: / by zero
	at westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:5)
```

Example:

```
public class Throw_Keyword {
	public static void main(String[] args) {
		throw new ArithmeticException("/ by zero");
		System.out.println("Hello...");
	}
}
```

Output:

```
CE: Exception in thread "main" java.lang.Error: Unresolved compilation problem: Unreachable code

	at westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:6)
```

Case 3:

We can use throw keyword only for throwable if we are trying to use for normal java object. We'll get compile time error saying incompatible types.

Example:

```
public class Throw_Keyword {
	public static void main(String[] args) {
		throw new Throw_Keyword();
	}
}
```

Output:

```
CE: Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	No exception of type Throw_Keyword can be thrown; an exception type must be a subclass of Throwable

	at westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:5)
```

Example:

```
public class Throw_Keyword extends RuntimeException{
	public static void main(String[] args) {
		throw new Throw_Keyword();
	}
}
```

Output:

```
RE: Exception in thread "main" westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword
	at westpac.com.au_Exception_11_Multiple_try_catch.Throw_Keyword.main(Throw_Keyword.java:5)
```

12. 'throws' keyword

In our program if there is possibilities of rising checked exception then compulsory we should handle the checked exception otherwise we'll get compile time error saying:

```
Unreported Exception XXX must be caught or declared to be thrown
```

Example 1:

```
public class Print_Writer {
	public static void main(String[] args) {
		PrintWriter printWriter=new PrintWriter("Ankur.txt");
		printWriter.print("Hello Ankur...");
	}

}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Unhandled exception type InterruptedException

	at Westpac.com.au_02_Print_Writer.Print_Writer.main(Print_Writer.java:8)
```

Example 2:

```
public class Print_Writer {
	public static void main(String[] args){
		Thread.sleep(10000);
	}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
```

```
Unhandled exception type InterruptedException

at Westpac.com.au_02_Print_Writer.Print_Writer.main(Print_Writer.java:8)
```

We can handle this compile time error by using following two ways:
1st Way: By using try catch.

```
public class Print_Writer {
	public static void main(String[] args){
		try {
			Thread.sleep(10000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

2nd Way: By using throws Keyword.
We can use throws keyword to delegate responsibility of exception handling to the caller (It may be another method or JVM). Then caller method is responsible to handle that exception.

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class Sleep_Method {
	public static void main(String[] args) throws FileNotFoundException {
		PrintWriter printWriter=new PrintWriter("Ankur.txt");
		printWriter.print("Hello Ankur...");
	}
}
```

Or

```
public class Print_Writer {
	public static void main(String[] args) throws InterruptedException{
			Thread.sleep(10000);
	}
}
```

Throws keyword required for only for checked exceptions and usage of throws keyword for unchecked exceptions. There is no use or impact. Throws keyword only to convenience compiler and usage of throws keyword doesn't prevent abnormal termination of the program.

Problem:

```
public class Test {

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

	}
	public static void doStuff(){
		doMoreStuff();
```

```
	}
	public static void doMoreStuff(){
		Thread.sleep(10000);
	}
}
```

Error:

```
CE: Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Unhandled exception type InterruptedException

	at Westpac.com.au_05_Test.Test.doMoreStuff(Test.java:16)
	at Westpac.com.au_05_Test.Test.doStuff(Test.java:13)
	at Westpac.com.au_05_Test.Test.main(Test.java:9)
```

Solution:

```
public class Test {

	public static void main(String[] args) throws InterruptedException{
		doStuff();
	}
	public static void doStuff() throws InterruptedException{
		doMoreStuff();
	}
	public static void doMoreStuff() throws InterruptedException{
		Thread.sleep(10000);
		System.out.println("Hello....");
	}
}
```

Output:

```
Hello....
```

In the above program if we remove at least one throws statement then the code wouldn't compile.

Throws Clause:

1. We can use to delegate responsibility of the exception to the caller(It may be method or JVM)
2. It is required only for checked exception and usage of throws keyword for unchecked exceptions there is no impact.
3. It is required only to convenient compiler and usage of throws does not prevent abnormal termination of program.

Case 1:

We can use throws keyword for methods and constructors but not for classes.

```
	public class Test throws Exception {
```

```
        Test() throws Exception{

        }

        public void m1() throws Exception{

        }
}
```

Case 2:

We can use throws keyword only for throwable types if we are trying to use for normal java classes then we'll get Compile Time Error saying: Incompatible types

```
public class Test{
        public static void main(String[] args) {

        }

        public void m1() throws Test{

        }
}
```

Error: Incompatible types found Test required.

Solution:

```
public class Test extends RuntimeException{

        public static void main(String[] args) {

        }
        public void m1() throws Test{

        }
}
```

Case 3:

Example:

```
public class Test extends RuntimeException{
        public static void main(String[] args) {
                throw new Exception();
        }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Unhandled exception type Exception

        at Westpac.com.au_05_Test_Case1.Test.main(Test.java:5)
```

Example:

```
public class Test extends RuntimeException{
	public static void main(String[] args) {
		throw new Error();
	}
}
```

Output:

```
Exception in thread "main" java.lang.Error
	at Westpac.com.au_05_Test_Case1.Test.main(Test.java:5)
```

Case 4:

```
public class Test extends RuntimeException{
	public static void main(String[] args) {
		try{
			System.out.println("Hello Ankur...");
		}catch(ArithmeticException e){

		}
	}
}
```

Output:

```
Hello Ankur...
```

Case 5:

```
public class Test {
	public static void main(String[] args) {
		try{
			System.out.println("Hello Ankur...");
		}catch(Exception e){

		}
	}
}
```

Output:

```
Hello Ankur...
```

Case 6:

```
import java.io.IOException;

public class Test {
	public static void main(String[] args) {
		try{
			System.out.println("Hello Ankur...");
		}catch(IOException e){

		}
	}
```

```
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Unreachable catch block for IOException. This exception is never thrown from the try statement body

	at Westpac.com.au_05_Test_Case1.Test.main(Test.java:9)
```

Case 7:

When we will get **`InterruptedException`** Exception If we are using sleep() method, wait() method, join() method then only we get **`InterruptedException.`**

In or program within the try block if there is no chance of rising the exception then we can't write catch block for that exception. Otherwise we'll get compile time error Saying Exception XXX is never thrown in body of corresponding try statement.
But this rule is applicable only for **`Fully Checked Exception.`**

```
public class Test {
	public static void main(String[] args) {
		try{
			System.out.println("Hello Ankur...");
		}catch(InterruptedException e){

		}
	}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Unreachable catch block for InterruptedException. This exception is never thrown from the try statement body

	at Westpac.com.au_05_Test_Case1.Test.main(Test.java:9)
```

Case 8:

```
package Westpac.com.au_05_Test_Case1;

import java.io.IOException;

public class Test {
	public static void main(String[] args) {
		try{
			System.out.println("Hello Ankur...");
		}catch(Error e){

		}
	}
}
```

Output:

```
Hello Ankur...
```

13. Exception Handling Keywords Summary

1. try ➔ To Maintain Risky Code
2. catch ➔To Maintain Exception Handling Code
3. finally ➔To Maintain Cleanup Code
4. throw ➔ To Handover our Created Exception Object to the JVM Manually.
5. Throws ➔ To Delegate Responsibility of Exceptional Handling to the Caller.

14. Various possible compile time errors in exception Handling

Various possible compile time error in Exceptional Handling is:

1. Unreported Exception XXX; must be caught or declared to be thrown
2. Exception XXX has already been caught
3. Exception XXX is never thrown in body of corresponding try statement.
4. Unreachable statement.
5. Incompatible type
   Found: Test
   Required: java.lang.Throwable
6. try without catch or finally
7. catch without try
8. finally without try

15. Customized or User Defined Exceptions

Some times to meet programming requirements we can define our own exception such type of exceptions are called Customized or user defined exceptions.
Example: we take the Example from Matrimonial Site. Like
ToYoungExcption
ToOldException
InSufficientFundsExceptions etc.

Note:

1. throw keyword is best suitable for use-defined or customized exception but not for pre-defined exceptions
2. It is highly recommended to define customized exception as unchecked i.e. we have to extends RuntimeException but not exception.
3. We use super(s); ➔ To Make Description Available to Default Exception Handler.

Example:

TooYoungException.java

```
public class TooYoungException extends RuntimeException {
	TooYoungException(String s){
		super(s);
	}
}
```

TooOldException.java

```
public class TooOldException extends RuntimeException {
	TooOldException(String s){
		super(s);
	}
```

```
}
```

CustExceptionDemo.java

```
public class CustExceptionDemo extends RuntimeException {
	public static void main(String[] args) {
		// int age=Integer.parseInt(args[0]);
		int age = 0;
		if(age>60){
			throw new TooOldException("Please wait some more time definitily
you will get best match");
		}
		else if(age<18){
			throw new TooYoungException("Your age already crossedmarriage age
no chance of getting merriage");
		}
		else{
			System.out.println("You'll get match details soon by Email!!! ");
		}
	}
}
```

16. Top -10 Exceptions

Based on the person who is rising the exception. All Exceptions are divided into 2 categories:

1- JVM Exception
2- Programmatic Exception

JVM Exceptions:
The exceptions which are raised automatically by JVM. Whenever a particular a event occurs are called JVM exceptions.
Example:
Arithmetic Exception, NullPointerException, etc.

Programmatic Exception:
The exceptions which are raised explicitly either by programmer or by API Developer to indicate that something goes wrong are called Programmatic Exceptions.
Example:
TooOldException, IllegleArgumentException, etc.

Top -10 Exceptions

1. **ArrayIndexOutOfBoundException:**
   1. It is the child class of Runtime Exception and Hence it is unchecked.
   2. Raised automatically by JVM whenever we are trying to access array element with out of Range Index.

   Example:
   int[] x=new int[4];
   sopln(x[0]);

   sopln(x[0]);
   RE:ArrayIndexOutOfBountException

   sopln(x[-10]);
   RE:ArrayIndexOutOfBountException

2. NullPointerException

1. It is the child class of RuntimeException and hence it is unchecked
2. Raised automatically by JVM whenever we are trying to perform any operation on null.

Example:

```
String s=null;
sopln(s.length());
```

RE:NullPointerException

3. ClassCastException:

1. It is the child class of RuntimeException and hence it is unchecked.
2. Raised automatically by JVM whenever we are trying to typecast parent object to child type.

Example:

```
String s=new String("Ankur");
Object o=(Object)s;
```

Output:Valid

```
Object o=new Object;
String s=(String)o;
```

Output:Invalid

RE:ClassCastException

```
Object o=new String("Ankur");
```

Output:Valid

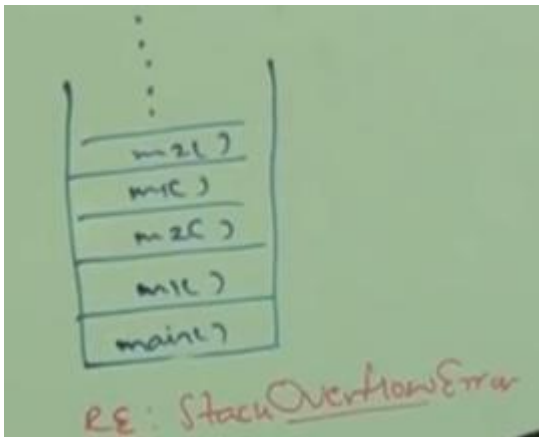
```
String s=(String)o;
```

Output:Valid

4. StackOverflowError:

1. It is the child class of Error and hence it is unchecked.
2. Raised automatically by JVM whenever we are trying to perform recursive method call.

```
class Test{
    p s v m1(){
            m2();
    }
    p s v m2(){
            m1();
    }
    p s v main(String[] args){
            m1();
    }
}
```



5. **NoClassDefFoundError**
    1. It is the child class of Error and hence it is unchecked.
    2. Raised automatically by JVM whenever JVM unable to find required .class file.

If Test.class file is not available then we'll get runtime Exception **NoClassDefFoundError:Test**

java Test ←

RE: NoClassDefFoundError: Test

6. **ExceptionInInitializerError:**
    1. It is the child class of Error and hence it is unchecked.
    2. Raised automatically by JVM if any exception occurs while executing static variable assignment and static blocks

Example 1:

```
public class Test {
	public static void main(String[] args) {
		static int x=10/0;
		System.out.println(x);
	}
}
```

Error:
ExceptionInInitializerError
caused By:java.lang.AE:/ by zero

Example 2:

```
public class Test {
	public static void main(String[] args) {
		static{
			String s=null;
			System.out.println(s.length);

		}
	}
```

```
}
Error:
ExceptionInInitializerError
caused By:java.lang.NullPointerException:
```

7. IllegleArgumentException:

1. It is the child class of RuntimeException and hence it is unchecked.
2. Raised explicitly either by programmer or by API developer to indicate that a method has been invoked with IllegalArgument.

Example:
The valid range thread priorities is 1-10 . If you ar etrying the set of priority with any other value then we'll get runtime exception saying IllegalArgumentException.

```
Thread t=new Thread();
t.setPriority(7);    // Valid
t.setPriority(15);  //InValid

RE:IllegalArgumentException
```

8. NumberFormatException

1. It is the direct child class of IllegalArgumentException which is the child class of RuntimeException and hence it is unchecked.
2. Raised explicitly either by programmer or by API developer to indicate that we are trying to convert String to Number and the String is not properly formatted.

```
int i=Integer.parseInt("10");
status:Valid
int i=Integer.parseInt("ten");
RE:NumberFormatException
status:Invalid
```

9. IllegalStateException

1. It is the child class of RuntimeException and hence it is unchecked.
2. Raised explicitly either by programmer or by API developer to indicate that a method has been invoked at wrong time.

Example:After stating of a thread we are not allow to restart the same thread once again.Otherwise we'll get Runtime Exception. IllegalThreadStateException.

```
Thread t=new Thread();
t.start();
   :
   :
   :
   :
t.start();
Error: IllegalThreadStateException.
```

10. AssertionError

1. It is the child class of Error and hence it is unchecked.
2. Raised explicitly by programmer or by API developer to indicate that assertStatmentFails .

if x is not greater than train then we'll get Runtime Saying AssertionError.

assert(x>10);

Exception/Error	Raised By
1. ArrayIndexOutOfBoundException	Raised Automatically by JVM and hence these are JVM Exceptions
2. NullPointerException	
3. ClassCastException	
4. StackOverFlowError	
5. NoClassDefFoundError	
6. ExceptionInInitializer	
7. IllegalArgumentException	Raised explicitly either by programmer or by API developer and hence these are programmatic exception.
8. NumberFormatException	
9. IllegalStateException	
10. AssertionError	

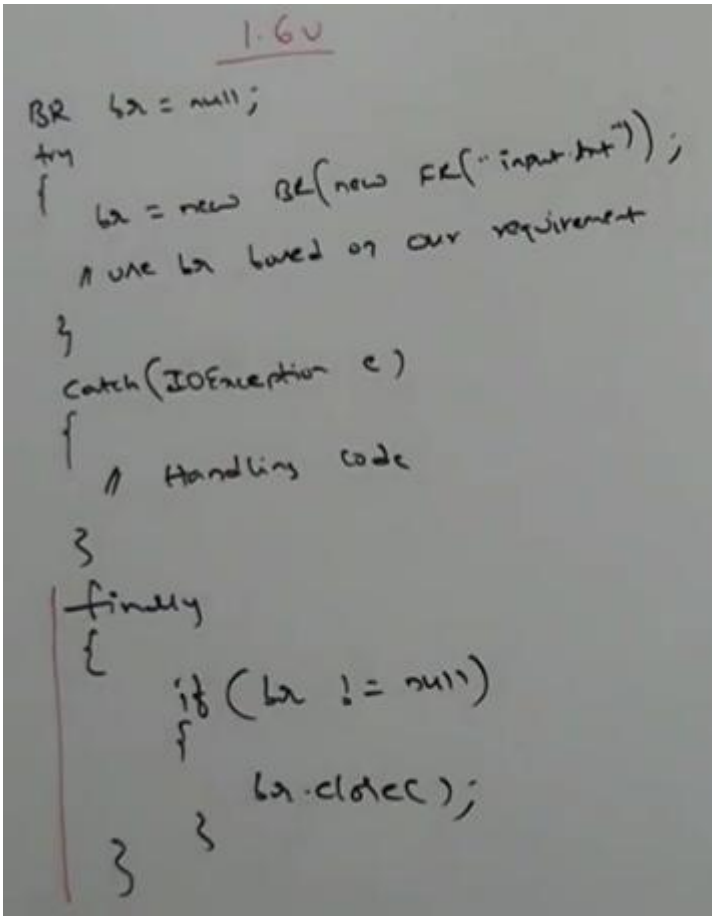
17. 1.7 version enhancements w.r.t Excetional Handling

As a part of 1.7 Version in exceptional handling the following 2 concept introduced:

1. Try with resources
2. Multi catch Block

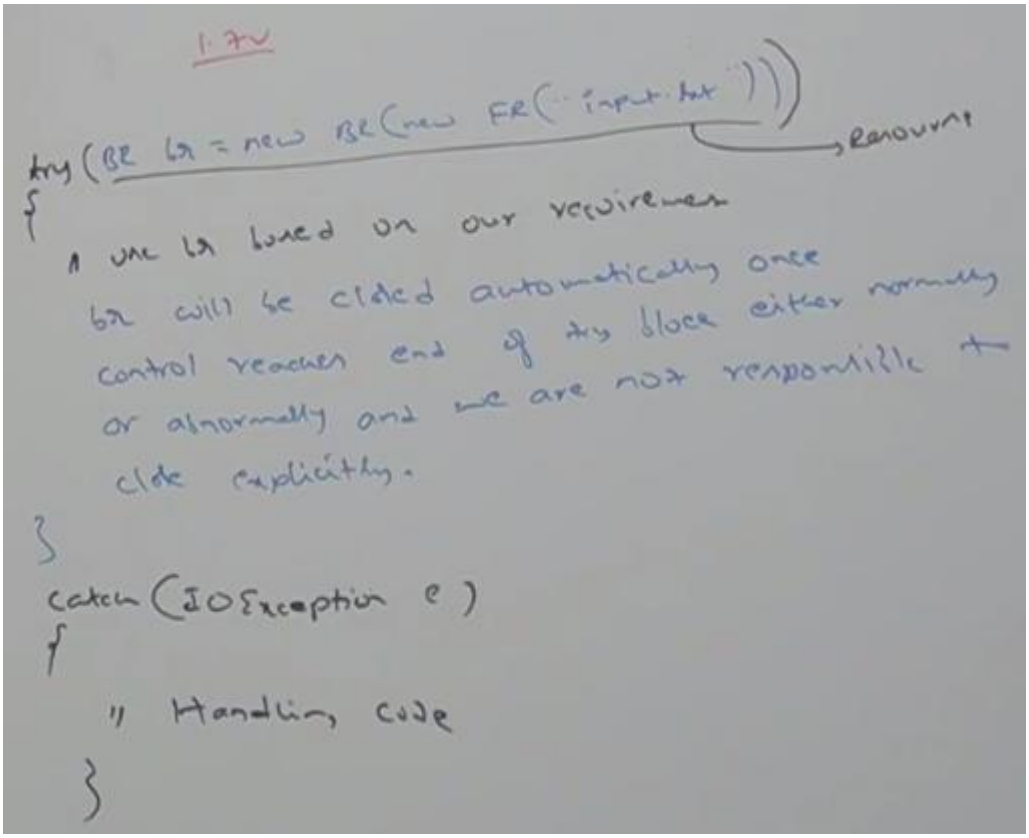
1. try with resources

Until 1.6 version it is highly recommended to write finally block to close resources. Which are open as a part of try block.



The problems in this approach are:

1- Compulsory programmer is required to close resources inside finally block. It increases complexity of programming.
2- We have to write finally block compulsory and hence it increases length of the code and reduces readability.
3- To overcome above problems SUN people introduced try with resources in 1.7 Version.
4- The main advantage of try with resources is whatever resources we open as a part of try block will be closed automatically once control reaches end of try block either normally or abnormally. And hence we are not required to close explicitly so that complexity of programming will be reduced.
5- We are not required to write finally block so that length of the code will be reduced and readability will be improved



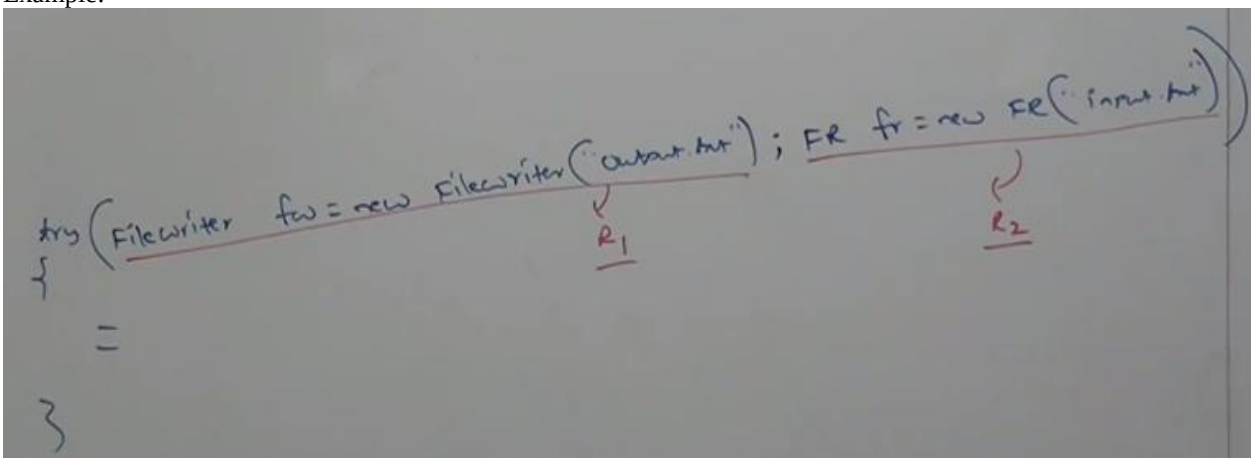
Conclusion 1:

We can declared multiple resources but these resources should be separated with Semi-Colon (;)

Syntax:

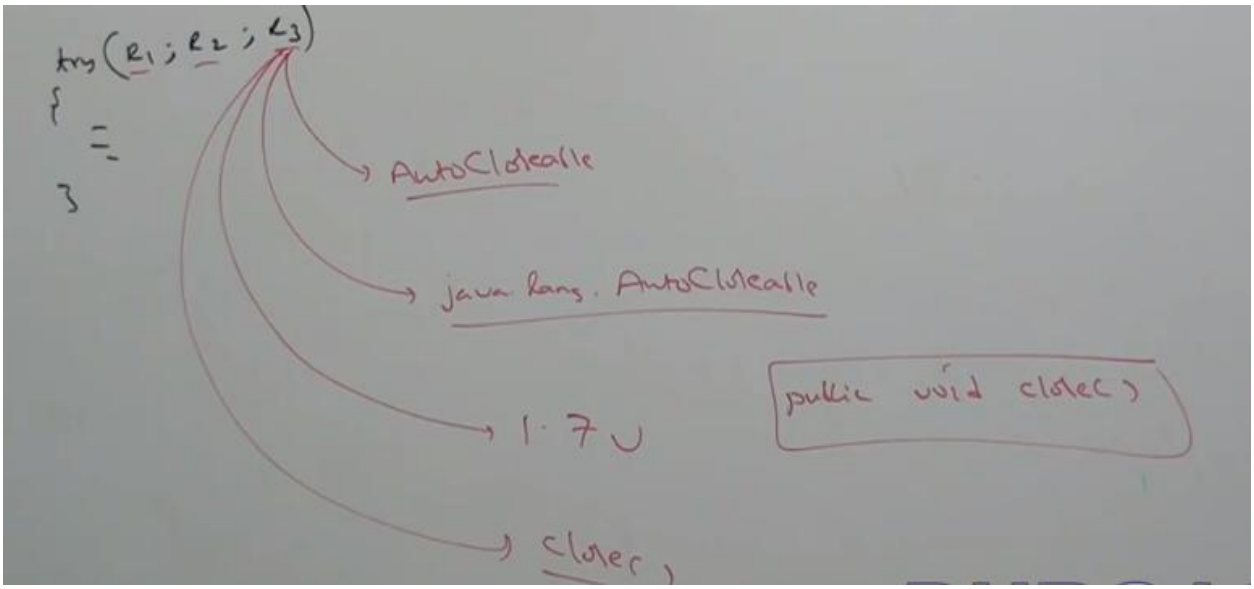
```
try(r1:r2:r3){
        ---
        ---
        ---
}
```

Example:



Conclusion 2:

- All resources should be auto-closable resources.
- A resources is said to be auto-closable if and only if corresponding class implement java.lang.AutoClosable Interface.
- All IO resources, database related Resources, And network related resources are already implemented auto closable interface.
- Being a programmer we are not required to do anything just we should aware the point.
- Auto-closable interface came in 1.7 Version. And it contains only one method close.



Conclusion 3:

All resource reference variable are implicitly final and Hence within the try block we can't perform re-assignmnet otherwise we'll get compile time error.

Example:

```
import java.io.BufferedReader;
import java.io.FileReader;

public class Try_with_Resources {
	public static void main(String[] args) {
		try(BufferedReader bufferedReader=new BufferedReader(new
FileReader("input.txt"))){
			bufferedReader=new BufferedReader(new FileReader("output.txt"));

		}catch(Exception e){

		}
	}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The resource bufferedReader of a try-with-resources statement cannot be assigned
```

```
	at Westpac.com.au_07_Try_with_Resources.Try_with_Resources.main(Try_with_Resources.java:9)
```

Example 2:

```
import java.io.BufferedReader;
import java.io.FileReader;

public class Try_with_Resources {
	public static void main(String[] args) {
		try(BufferedReader bufferedReader=new BufferedReader(new
FileReader("input.txt"))){
			//bufferedReader=new BufferedReader(new
FileReader("output.txt"));
		}catch(Exception e){

		}
	}
}
```

Note:

- Until 1.6 version try should be associated with either catch or finally. But from 1.7 version onward we can take only try with resource without catch or finally.

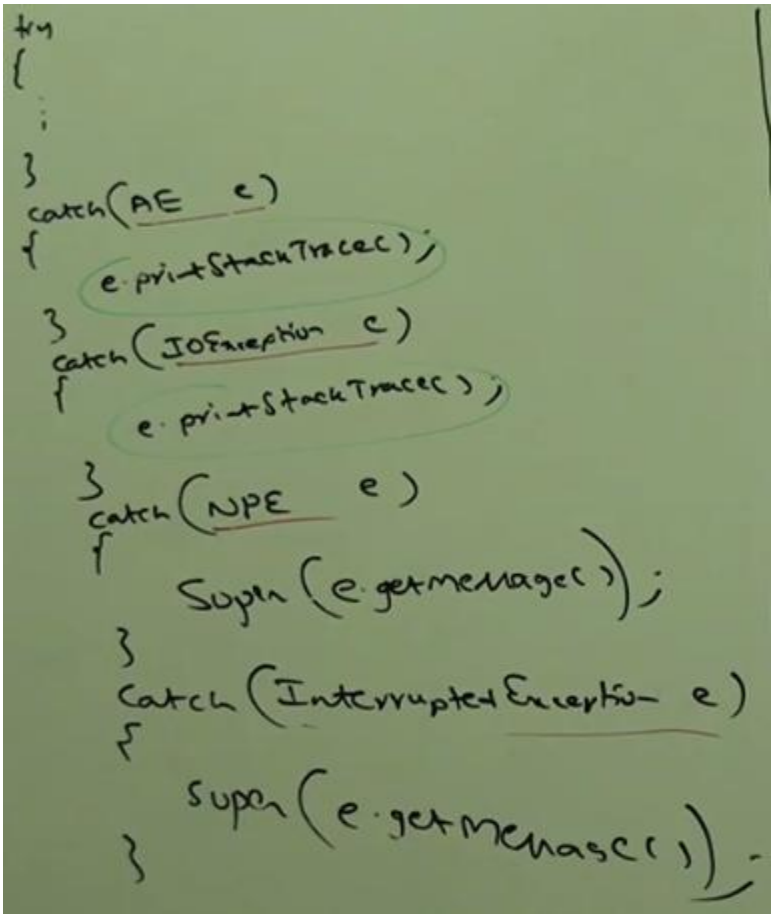
  ```
  try(R){

  }
  ```

- The main advantage of try with resources is we are not required to write finally block explicitly because we are not required to close resources explicitly. Hence until 1.6 version finally bock is just like Hero but 1.7 version onward it is dummy and becomes Zero.

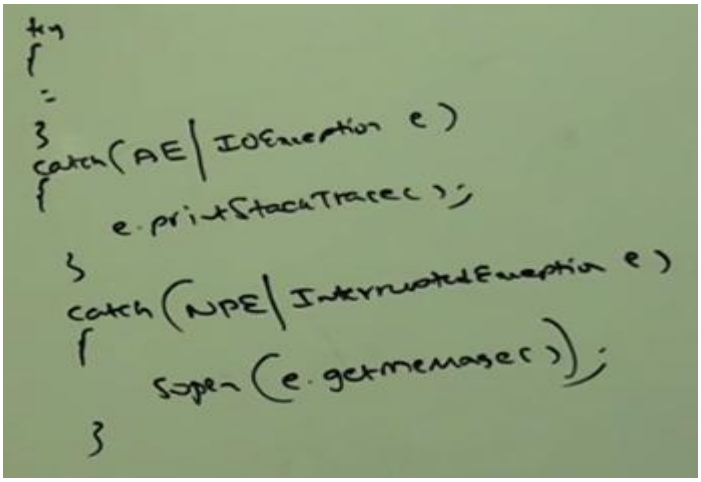
2. Multi-catch Block

Until 1.6 Version even though multiple different exception having same handing code for every exception type we have to write a separate catch block. It increases length of the code and reduces readability



- To overcome this problem Sun people introduced Multi-Catch Block in 1.7 version.
- According to this we can write a single catch block that can handle multiple different type of exception.

- The main advantage of this approach is length of the code will be reduced and readability will be improved.



Example:

```
public class Try_with_Resources {
	public static void main(String[] args) {
		try{
			System.out.println(10/0);
			String s=null;
			System.out.println(s.length());
		}catch(ArithmeticException|NullPointerException e){
			System.out.println(e);
		}
	}
}
```

Output:

```
java.lang.ArithmeticException: / by zero
```

In the above example whether raised the exception is either arithmeticException or NullPointerException the same catch block can respond.

- In Multi-catch block there should not be any relation between exception types (Either child to parent or parent to child or same type) otherwise we'll get compile time error.

Example:

```
public class Try_with_Resources {
	public static void main(String[] args) {
		try{
			System.out.println(10/0);
			String s=null;
			System.out.println(s.length());
		}catch(ArithmeticException|Exception e){
			System.out.println(e);
		}
	}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The exception ArithmeticException is already caught by the alternative Exception

	at
Westpac.com.au_07_Try_with_Resources.Try_with_Resources.main(Try with Resources.java:9)
```

Exception Propagation:

Inside a method if an exception raised and if we are not handling that exception then exception object will be propagated to caller then caller method is responsible to handle exception. This process is called exception propagation.

Re- Throwing Exception:
We can use this approach to convert one exception to another exception

```
		try{
			System.out.println(10/0);
		}catch(ArithmeticException e){
			throw new NullPointerException();
}
```

18. Difference between ClassNotFoundException and NoClassDefFoundError:

Case 1:

Test t=new Test();
If runtime Test.class file is not available then that time we get error as a NoClassDefFoundError
RE:NoClassDefFoundError
This is unchecked Exception.

Case 2:

Object o=Class.forname(args[0]).newInstance()
Java Test Student ➔

For dynamically provided class name , at runtime if the corresponding .class file is not availble then we'll get runtime exception like ClassNotFoundException.
RE:ClassNotFoundException
This is checked Exception.