Exception Handling

Module 6: Exception Handling	
1.	Intro & Runtime Stack mechanism
2.	Default exception handling in java
3.	Exception Hierarchy
4.	Customized exception handling by using try-catch
5.	Control flow in try-catch, try-catch-finally
6.	Methods to print exception information
7.	Tyr with multiple catch blocks
8.	final vs finally vs finalize
9.	throw vs throws keyword
10.	Various Compile time errors
11.	Customized or user defined exceptions
12.	Top 10 exceptions
13.	Try with resources, multi-catch block
14.	Operator precedence & Evaluation order of operands

1. Intro & Runtime Stack mechanism

- An unexpected unwanted event that disturbs Normal flow of the program is called Exception.
  E.g., TyrePunturedException, SleepiingException, FileNotFoundException etc.

- It is highly recommended to handle Exception & the main objective of Exception handling is graceful termination of the program.
- Exception handling doesn't mean repairing an exception. We have to provide alternative way to continue rest of the program normally is the concept of Exception Handling.

  E.g., Our programming requirement is to Read data from remote file located at London. At Runtime, if London file is not available, our program should not be terminated abnormally, there must be some alternative local file to continue rest of the program normally.

```
try {
    Read data from remote file locating at london
} catch (FileNotFoundException e) {
    Use localfile & continue rest of the program normally
}
```

- For every Thread, JVM will create a Runtime Stack
  a) Each & every method call performed by that thread will be stored in the corresponding thread.
  b) Each entry in the stack is called Stack frame / Activation record.
  c) After completing every method call, the corresponding entry from stack will be removed.
  d) After completing all method call, the stack will become empty & that empty stack will be destroyed by JVM just before terminating the thread.

```
Class Test {

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

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


  public static void doMoreStuff () {
     SoPln("Hello");
  }

}

Output: Hello
```

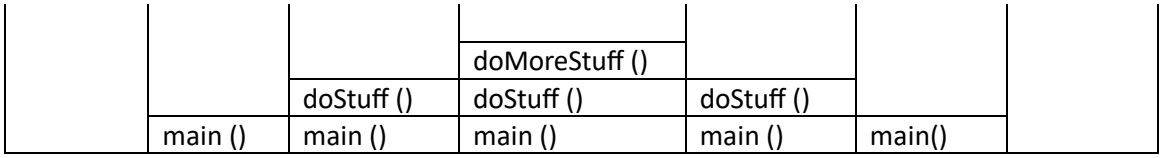


Fig: Runtime Stack

2. Default Exception handling in Java

Step 1: Inside a method if any exception occurs, then that method is responsible to create exception object by including the following information

- Name of exception
- Description of exception
- Location at which exception occurs [Stack Trace]

Step 2: After creating exception object, method hand over the object to JVM.

Step 3: JVM will check whether the method contains any exception handling code or not. If that method doesn't contain exception handling code, then JVM terminates that method abnormally & removes the corresponding entry from the stack. Then it will continue checking all the caller method for exception handling code until main () method.

Step 4: If the main () method also doesn't contain exception handling code then JVM terminates main () method abnormally & removes corresponding entry from the stack.

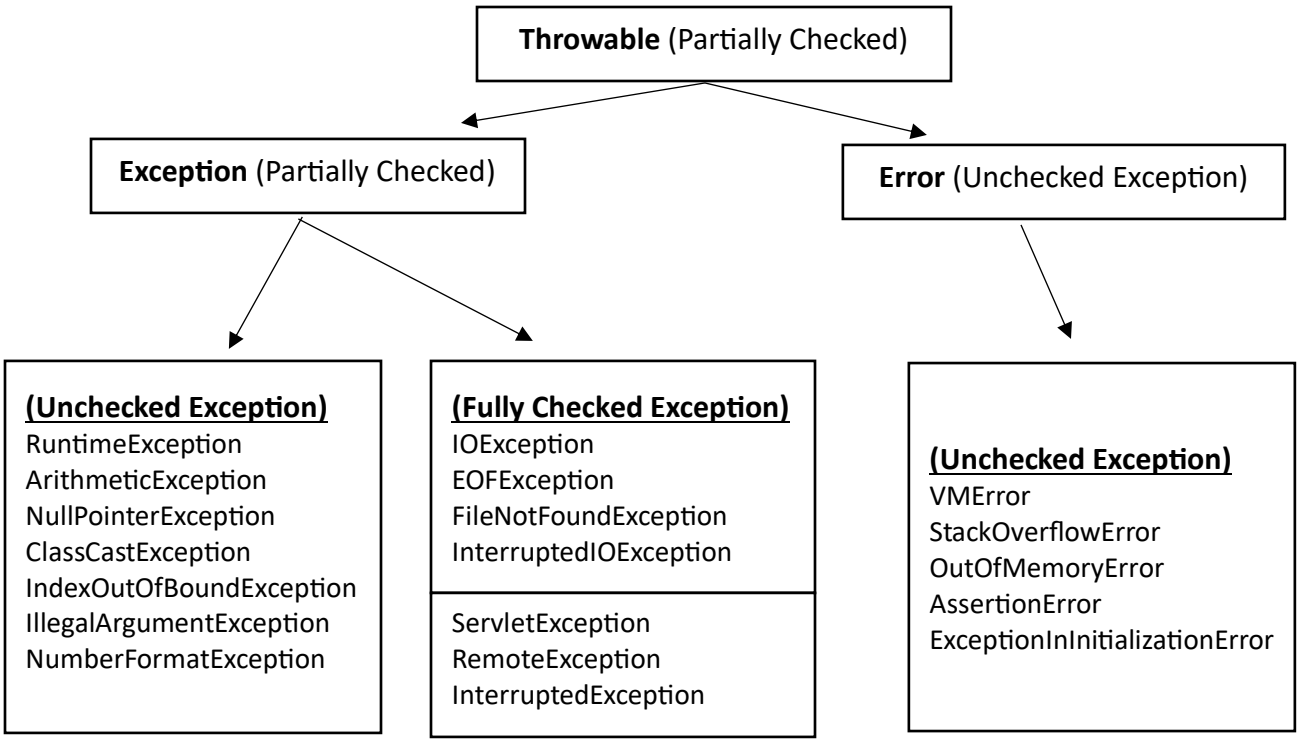
Step 5: Then JVM hands over responsibility of exception handling to the default exception handler which is the part of JVM. Default Exception Handler prints exception information in the following format & terminates program abnormally.

Exception in thread "xxxx" Name of exception: Description, Stack Trace

- In a program, if at least one method terminates abnormally then the program termination is Abnormal termination.
- If all methods terminated normally, then only Program termination is Normal Termination.

3. Exception Hierarchy

- **Throwable** class acts as Root for Java exception Hierarchy.
- Throwable class defines 2 child classes: **Exception** & **Error**.
- **Exceptions**: Most of the times, exception are caused by our program & these are recoverable. Check above example.
- **Error**: Most of the times, errors are not caused by our program & these are due to lack of system resources. Errors are non – recoverable. For e.g., OutofMemoryError etc.



Checked Exception Vs Unchecked Exception

No.	Checked Exception	Unchecked Exception
1.	The exceptions which are checked by compiler for smooth execution of the program is called Checked Exception. 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 will get Compile Time Error.	The exceptions which are not checked by compiler whether programmer handling or not, such type of exceptions are called Unchecked exception.
2.	FileNotFoundException, IOException etc.	ArithmeticException, NullPointerException etc.
	Fully Checked Exception	**Partially Checked Exception**
1.	A Checked exception is said to be fully checked iff all its child classes also checked.	A Checked exception is said to be partially checked iff some of its child classes are unchecked.

4. Customized exception handling by using try – catch

- It is highly recommended to handle exceptions.
- The code which may rise an exception is called Risky code & we have to define that code inside try block & corresponding handling code we have to define inside catch block.

```
try {
   // Risky code
} catch (Exception e) {
   // Handling code
}
```

Without try – catch	With try – catch

class Test {
  Public static void main (String [] args) {
  SoPln (“Stat1”);
  SoPln (10/0);
  SoPln (“Stat3”);
  }
}
 	
Output:
Stat1
RuntimeException: AE: / by zero
	class Test {
	  Public static void main (String [] args) {
	  SoPln (“Stat1”);
	  try {
	  SoPln (10/0);
	  } catch (ArithmeticException e) {
	  SoPln(“Stat2”);
	  }
	  SoPln (“Stat3”);
	  }
	}
	
	Output:
	Stat1
	Stat2
	Stat3

- Within try block, if anywhere an exception raised then rest of the try block won’t be executed even though we handled that exception, hence within the try block, we have to take only risky code & length of try block should be as less as possible.
- In addition to try block, there may be a chance of raising an exception inside catch & finally blocks.
- If any statement which is not part of try block raises an exception, then it’s always abnormal termination.

5. Methods to print exception information

- Throwable class defines the following methods to print exception information
- Internally Default exception handler will use printStackTrace () method to print exception information to the console.

No.	Method	Printable Format	Example
1.	printStackTrace ()	Name of Exception: Description, Stack Trace	e.printStackTrace ()
2.	toString ()	Name of Exception: Description	SoPln(e) or SoPln (e.toString ())
3.	getMessage ()	Description	SoPln (e.getMessage ())

6. try with multiple catch blocks

- The way of handling an exception is varied from exception to exception, hence for every exception type, its highly recommended to take separate catch block i.e., try with multiple catch blocks is always possible & recommended to use.

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

- We can't declare 2 catch blocks for the same exception we will get Compile time error.

```
try {
        // Risky Code
} catch (ArithmeticException e) {
       // Perform alternative arithmetic operations
} catch (SQLException e) {
      // Use myself db instead of Oracle db
} catch (FileNotFoundException e) {
     // Use local file instead of remote file
} catch (Exception e) {
    // default exception handling
}
```

```
// Wrong Order of Exception handling
try {
        // Risky Code
} catch (Exception e) {
    // default exception handling
} catch (ArithmeticException e) {
       // Perform alternative arithmetic operations
}

// Correct order of Exception handling
try {
        // Risky Code
} catch (ArithmeticException e) {
       // Perform alternative arithmetic operations
} catch (Exception e) {
    // default exception handling
}
```

7. final Vs finally Vs finalize

No.	final	finally	finalize
1.	final is a modifier applicable for classes, methods & variables.	finally is a block always associated with try – catch to maintain cleanup code.	finalize () is a method always invoked by Garbage collector just before destroying an object to perform cleanup activities.

2.	If a class is final then we can't extend that class.
	
	If a method is final, then we can't override that method in the child class.
	
	If a variable is final, we can't perform re – assignment for that variable.
		The specialty of finally block is it will be executed always irrespective of whether exception is raised or not & whether exception handled or not.
		
		try {
		// Risky Code
		} **catch** (Exception e) {
		// Handling Code
		} **finally** {
		// Cleanup Code
		}
			Once finalize () method completes, immediately Garbage collector destroys that object.

3.		finally block is responsible to perform clean activities related to try block i.e., whatever resources we opened in try block will be closed inside finally block.	finalize () method is responsible to perform cleanup activities related to Object i.e., whatever resources associated with object will be deallocated before destroying an object by using finalize() method.

8. throw Keyword

- Sometimes we can create exception object explicitly & we can hand over to JVM manually. For this we have to use "throw" keyword

throw new Arithmetic Exception ("/ by zero");

throw – it hands over our created exception object to the JVM manually
new ArithmeticException () – Creation of ArithmeticException object

- Hence, the main objective of throw keyword is to hand over our created exception object to the JVM manually.
- Hence, the result of the following 2 programs is exactly same. The only diff is how exception object will be handed over to the JVM.
- Best use of throw keyword is for user – defined exceptions or Customized exception.

Without throw keyword	With throw keyword

class Test {
public static void main (String [] args) {
SoPln (10/0);
}
}
 	
RE: Exception in thread "main"
java.lang.ArithmeticException: / by zero at Test.main
	class Test {
	public static void main (String [] args) {
	throw new ArithmeticException ("/ by zero");
	}
	}
	
	RE: Exception in thread "main"
	java.lang.ArithmeticException: / by zero at Test.main

In this case, main () method is responsible to create exception object & hand over to the JVM.	In this case, Programmer is creating exception object explicitly & hand over to the JVM manually.

- **Case 1:** throw e; (If e refers to null then we will get NullPointerException)
- **Case 2:** After throw statement, we are not allowed to write any statement directly otherwise we will get **CE saying Unreachable statement.**
- **Case 3:** We can use throw keyword only for Throwable types. If we're trying to use for normal java Objects, we will get **CE: incompatible types.**

9. 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.

- In our program, if there is a possibility of raising Checked Exception then compulsory, we should handle that checked exception otherwise we will get CE:

```
CE: Unreported exception xxxx; must be caught or declared to be thrown
```

- We can handle these compile time error by using the following 2 ways:
  a) By using try – catch
  b) By using throws keyword

Imp. Points for throws keyword

- throws keyword is required only for checked exceptions & usage of throws keyword for unchecked exception there is no impact.
- Its required only to convince compiler & usage of throws doesn't prevent abnormal termination of the program.
- It is recommended to use try – catch over throws keyword.

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

Case 1: We can use throws keyword for methods & constructors but not for classes.

```
class Test {
   Test () throws Exception {}
   public void m1 () throws Exception {}
}
```

Case 2: We can use throws keywork only for Throwable types. If we're trying to use for normal java classes then we will get CE: incompatible types.

// Wrong	// Correct

class Test {
  public void m1 () throws Test {}
}
 	
CE: incompatible type
found: Test
required: java.lang.Throwable
	class Test extends RuntimeException {
	  public void m1 () throws Test {}
	}

Case 3:

class Test {
  public static void main (String [] args) {
  throw new Exception ();
  }
}
 	
CE: unreported exception Exception; must be caught or declared to be thrown
	class Test {
	  public static void main (String [] args) {
	  throw new Error ();
	  }
	}
	
	RE: Exception in thread "main" ERROR! Java.lang.Error at Test.main()

10. Customized or User – defined exceptions

- Sometimes to meet programming requirements we can define our exceptions; such type of exceptions are called customized or user – defined exceptions.
- "throw" keyword is best suitable for user – defined or customized exceptions but not for Pre – defined exceptions.
- It is highly recommended to define customized exceptions as Unchecked exception i.e., we have to extend RuntimeException but not Exception.
- Inside every customized exception constructor, we're taking super () to make description available to the default exception handler which then use printStackTrace () method to print exception.

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

// RuntimeException is Unchecked exception

// Using super () to make description available to default exception handler
```

11. Top 10 Exceptions

a) JVM Exceptions – Exceptions raised automatically by JVM whenever a particular event occurs.
e.g., ArrayIndexOutofBoundException, NullPointerException, ClassCastException, StackOverflowError, NoClassDefFoundError, ExceptionInInitializationError

b) Programmatic Exceptions – Exceptions raised explicitly either by Programmer or API developer to indicate that something goes wrong
e.g., IllegalArgumentException, NumberFormatException, IllegalStateException, AssertionError

No.	Exception	Description
1.	ArrayIndexOutOfBoundException	• It is the child class of RuntimeException & Unchecked • Raised automatically by JVM whenever we're trying to access array element with out-of-Range index.
2.	NullPointerException	• It is the child class of RuntimeException & Unchecked • Raised automatically by JVM whenever we're trying to perform any operation on null.
3.	ClassCastException	• It is the child class of RuntimeException & Unchecked • Raised automatically by JVM whenever we're trying to typecast parent object to child type.
4.	StackOverflowError	• It is the child class of Error & Unchecked • Raised automatically by JVM whenever we're trying to perform recursive method call.

5.	NoClassDefFoundError	• It is the child class of Error & Unchecked • Raised automatically by JVM whenever JVM unable to find required .class file

6.	ExceptionInInitializerError	• It is the child class of Error & Unchecked
		• Raised automatically by JVM if any exception occurs while executing static variable assignments & static blocks.
		e.g.,
		class Test {
		static int x = 10/0;
		}
		RE: ExceptionInInitializerError caused by AE: / by zero.

7.	IllegalArgumentException	• It is the child class of RuntimeException & Unchecked
		• Raised explicitly either by Programmer or by API developer to indicate that a method has been invoked with illegal argument.
		e.g., Thread t = new Thread ();
		t.setPriority(7);
		t.setPriority(15); // RE: IllegalArgumentException

8.	NumberFormatException	• It is the direct child class of IllegalArgumentException & Unchecked.
		• Raised explicitly either by Programmer or by API developer to indicate that we're trying to convert String to Number & the String is not properly formatted.
		e.g., int I = Integer.parseInt("10");
		int I = Integer.parseInt("ten"); // RE: NumberFormatException

9.	IllegalStateException	• It is the child class of RuntimeException & Unchecked
		• Raised explicitly either by Programmer or by API developer to indicate that a method has been invoked at wrong time.
		
		e.g., After starting off a thread, we're not allowed restart same thread once again otherwise we will get RE: IllegalStateException
		Thread t = new Thread ();
		t.start();
		…
		t.start(); // RE: IllegalStateException

10.	AssertionError	• It is the child class of Error & Unchecked • Raised explicitly either by Programmer or by API developer to indicate that assert statement fails. e.g., assert (x >10); // If x is not greater than 10, we will get RE: AssertionError

12. try with resources (Enhancement)

- Until 1.6v, it's highly recommended to write "finally" block to close resources which are opened as a part of try block.

```
BufferedReader br = null;
try {
    br = new BufferedReader (new FileReader("input.txt"));
} catch (IOException e) {
    // Handling code
} finally {
    if (br != null)
        br.close();
}
```

- The problems in this approach are:
  1. Programmer explicitly required to close resources inside finally block. It increases complexity of programming.
  2. Since finally block is compulsory; it increases length of the code & reduces readability.

- To overcome above problems, SUN people introduced try with resources in 1.7v
- The main advantage of try with resources is whatever resources we opened as part of try block will be closed automatically once control reaches end of try block either normally or abnormally & hence, we're not required to close explicitly so that complexity of programming will be reduced.

```
BufferedReader br = null;
try (br = new BufferedReader (new FileReader("input.txt"))) {
    // code
} catch (IOException e) {
    // Handling code
}
```

br will be closed automatically once control reaches end of try block either normally or abnormally & we are not responsible to close explicitly.

Imp. Point:

1. We can declare multiple resources but these resources should be separated with semicolon (;)
2. All Resources should be AutoCloseable resources. A Resource is said to be **AutoCloseable** iff corresponding class implements java.lang.AutoCloseable interface.
   E.g., All IO related resources, DB related resources & Network related resources are already implemented AutoCloseable interface.

   AutoCloseable interface contains only 1 method i.e., **public void close ();**

```
try (R1; R2; R3) {
    // code
} catch (IOException e) {
    // Handling code
}
```

3. All Resource reference variables are implicitly final & hence within the try block we can't perform a re – assignment otherwise we will get CE: auto – closeable resource br may be assigned.
4. Until 1.6v, try should be associated with either catch or finally but from 1.7v onwards, we can have only try with resources (without catch or finally)

13. Multi – catch block (Enhancement)

- Until 1.6v, multiple different exceptions having same handling code for every exception type, we have to write a separate catch block. It increases length of the code & reduces readability.

- To overcome this problem, SUN people introduced multi – catch block in 1.7v. According to this, we can write a single catch block that can handle multiple different type of exceptions.

```
try {
    // code
} catch (ArithmeticException | IOException e) {
    e.printStackTrace ();
} catch (NullPointerException | InterruptedException e) {
    SoPln (e.getMessage ());
}
```

- In multi catch block, there should not be any relation b/w Exception types (either child to parent / parent to child / same type) otherwise we will get CE: Alternatives in a multi – catch statement can't be related by subclassing.

- The main adv. of this approach is length of code will be reduced & readability will be improved.