How do we define exception?

Any unwanted unexpected event that interrupts normal flow execution.

What is the default exception handler?

Default exception is the handler provided by JVM. If no proper exception handling is done then JVM terminates all methods abnormally and terminates the main thread with the help of default exception handler.

```
public class MyException {
    public static void main(String[] args) {
        m1();
    }
    public static void m1() {
        m2();
    }
    public static void m2() {
        int i = 1/0;
    }
}
```

O/P:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at MyException.m2(MyException.java:14)
    at MyException.m1(MyException.java:10)
    at MyException.main(MyException.java:6)
```

Which class is the root of all exception and error class hierarchy?

Throwable class is the root of all exceptions and errors.

What are the immediate two subclasses of Throwable?

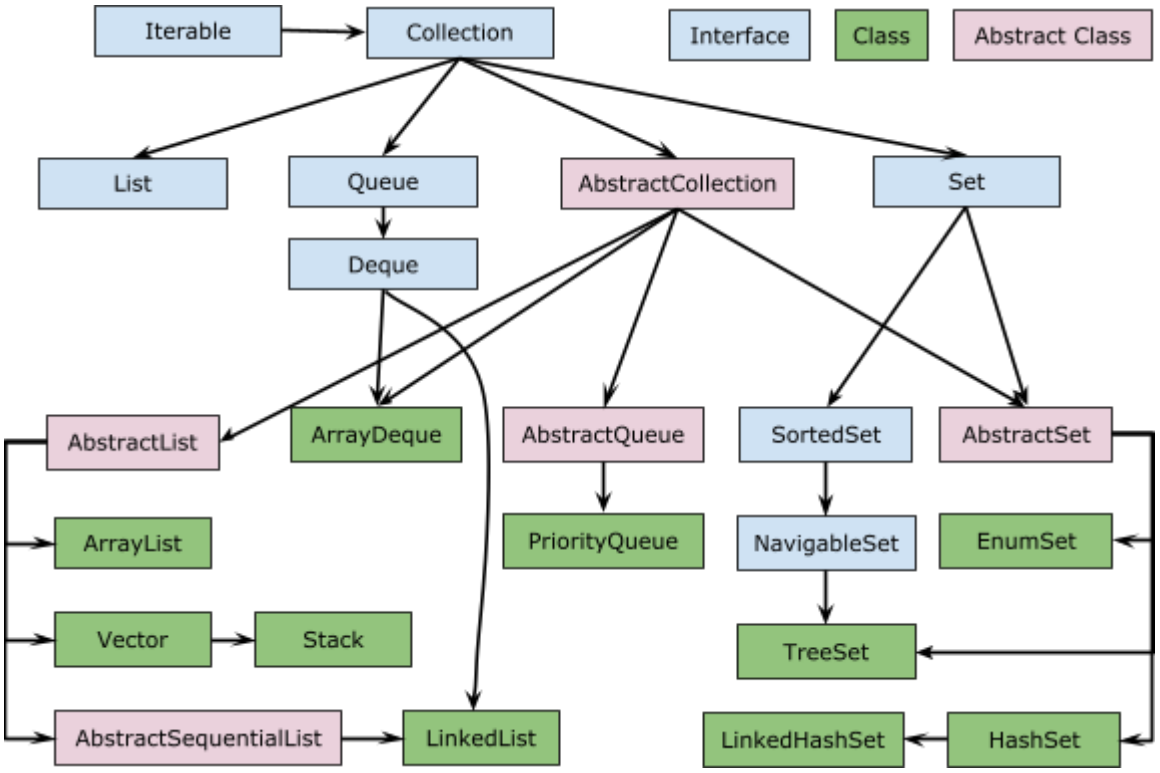
Exception and Error are the two immediate subclasses of Throwable.

What is the difference between Exception and Error?

Exception	Error
Recoverable: We can handle exception and add alternate conditions to end our	Non-recoverable: We cannot handle errors, when it occurs, it breaks the flow

program normally.	execution.
Exception are caused by the program only.	Error can be caused by a program, but most of the error occurs due to lack of system resources.

Hierarchy?



Difference between Exception and Error?

Throwable (1.0)
- -> Exception (1.0)
  - -> RuntimeException (1.0)
    - -> ArithmeticException (1.0) - ***
    - -> IndexOutOfBoundsException (1.0)
      - -> ArrayIndexOutOfBoundsException (1.0)
      - -> StringIndexOutOfBoundsException (1.0)
    - -> ArrayStoreException (1.0)
    - -> ClassCastException (1.0)
    - -> EnumConstantNotPresentException (1.5)
    - -> IllegalArgumentException (1.0)
      - -> IllegalThreadStateException (1.0)

- -> NumberFormatException (1.0)
- -> IllegalMonitorStateException (1.0)
- -> IllegalStateException (1.1)
- -> NullPointerException (1.0) - ***
- -> SecurityException (1.0)
- -> TypeNotPresentException (1.5)
- -> UnsupportedOperationException (1.2)
- -> ReflectiveOperationException (1.7)
  - -> ClassNotFoundException (1.0)
  - -> IllegalAccessException (1.0)
  - -> InstantiationException (1.0)
  - -> NoSuchFieldException (1.1)
- -> CloneNotSupportedException (1.0)
- -> InterruptedException (1.0)

-> Error (1.0)
- -> ThreadDeath (1.0)
- -> AssertionError (1.4)
- -> LinkageError (1.0)
  - -> BootstrapMethodError (1.7)
  - -> ClassCircularityError (1.0)
  - -> ClassFormatError (1.0) - ***
    - -> UnsupportedClassVersionError (1.2)
  - -> ExceptionInInitializerError (1.1)
  - -> IncompatibleClassChangeError (1.0)
    - -> IllegalAccessError (1.0)
    - -> NoSuchFieldError (1.0)
    - -> InstantiationError (1.0)
    - -> NoSuchMethodError (1.0)
  - -> NoClassDefFoundError (1.0)
  - -> UnsatisfiedLinkError (1.0)
  - -> VerifyError (1.0)
- -> VirtualMachineError (1.0)
  - -> InternalError (1.0)
  - -> OutOfMemoryError (1.0)
  - -> StackOverflowError (1.0)
  - -> UnknownError (1.0)

Which type of exception occurs at compile time, checked or unchecked?

Both checked and unchecked occur at run time only. At compile time, we only get syntax errors which we call compile time errors.

Difference between checked and unchecked exception?

Checked	Unchecked
Exceptions checked by the compiler during the compilation process. Developers must handle code and throw exception using throws keyword.	Exceptions which are not checked by the compiler.
FileNotFound,Intertu	ArithmeticException
Exception Error and Runtime exceptions all are checked.	- All Errors are unchecked - Runtime exception and child classes are unchecked

Checked exception example 1:

```
import java.io.FileReader;
import java.io.Reader;

public class MyReader {
   public static void main(String[] args) {
      Reader reader = new FileReader("test.txt");
   }
}
```

Output when trying to compile code:
MyReader.java:8: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown
Reader reader = new FileReader("test.txt");

Checked exception example 2:

```
public class MyThread {
   public static void main(String[] args) {
      Thread.sleep(1000);
   }
}
```

MyThread.java:3: unreported exception java.lang.InterruptedException; must be caught or declared to be thrown
Thread.sleep(1000);

What is the difference between fully checked and partially checked exception?

Fully Checked	Partially Checked
If the parent and all its child classes are checked by the compiler then it is called fully checked.	If a parent is checked but some or all of its child classes are unchecked then it is called partially checked.

Example: IOException	Example: Exception (IOException is checked, RuntimeException is unchecked). Only Throwable and Exception are partially checked.

How many methods are there in Throwable class?

1. getMessage

   ```
   public String getMessage() {
      return detailMessage;
   }
   ```

   Here detailMessage contains specific details about the Throwable. For example, for FileNotFoundException, this contains the name of the file that could not be found.

2. getLocalizedMessage

   ```
   public String getLocalizedMessage() {
      return getMessage();
   }
   ```

3.

the general purpose exception-chaining mechanism.

objects may be constructed by the virtual
 * machine as if {@linkplain Throwable#Throwable(String, Throwable,
 * boolean, boolean) suppression were disabled and/or the stack trace was not
 * writable}.

public final class StackTraceElement implements java.io.Serializable {

java.lang.SuppressWarnings

==

Test.java

```
public class Test {
   public static void main (String arr[]) {
         System.out.println("In main method of Test class");
   }
```

}

javac Test.java

Edit Test.class and remove last two character and run:
java Test

java Test
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.ClassFormatError: Truncated class file
	at java.lang.ClassLoader.defineClass1(Native Method)
	at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
	at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
	at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
	at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
	at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
	at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
	at java.security.AccessController.doPrivileged(Native Method)
	at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
	at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
	at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
	at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
	at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:495)

Unknown constant tag 108 in class file Test
Incompatible magic value 4273651200 in class file Test
Illegal UTF8 string in constant pool in class file Test
LineNumberTable attribute has wrong length in class file Test

==

NullPointerException occurs when to we try to perform some operation by considering it as java object.

Ex:
1. While executing some method on null
2. While accessing some member variable on null
3. Trying to get length of null

```
public class NullPointer {
	public String msg;
	public static void main(String[] args) {
	NullPointer obj = null;
	String str = null;
	try {
	System.out.println(obj.msg);
	} catch (Exception ex) {
	ex.printStackTrace();
	}
	try {
	System.out.println(str.length());
```

```
        } catch (Exception ex) {
        ex.printStackTrace();
        }
        System.out.println("In main method of NullPointer class");
        }
}
```

O/P:

```
java.lang.NullPointerException
        at NullPointer.main(Null.java:7)
java.lang.NullPointerException
        at NullPointer.main(Null.java:12)
In main method of NullPointer class
```

Basic Runtime constructor (2)

==

ArithmeticException occurs when we try to perform some arithmetic operation which is incorrect.

Ex:

1. While dividing zero by any number. (10/0)
2. While getting modulo of any number by zero. (10 % 0)

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

O/P:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Arithmetic.main(Arithmetic.java:4)
```

Basic Runtime constructor (2)

==

Do we always get an ArithmeticException while performing an operation divide by zero?

float or double

```
public static void main(String[] args) {
   float input = 0;
   System.out.println(10/input);
}
```

O/P:

```
Infinity
```

==

ArrayIndexOutOfBoundsException occurs when we try to perform some illegal index of an array.

Ex:
1. Accessing index which is not in range of that array.
2. Assigning value to index greater than size of that array.

```
public class ArrayIndexOutOfBounds {
	public static void main(String[] args) {
	String[] arr = new String[2];
	arr[0] = "Test1";
	arr[1] = "Test2";
	try {
	arr[2] = "Test3";
	} catch (Exception ex) {
	ex.printStackTrace();
	}
	try {
	System.out.println(arr[3]);
	} catch (Exception ex) {
	ex.printStackTrace();
	}
	}
}
```

O/P:

```
java.lang.ArrayIndexOutOfBoundsException: 2
	at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:7)
java.lang.ArrayIndexOutOfBoundsException: 3
	at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:12)
```

==

StringIndexOutOfBoundsException is also similar to ArrayIndexOutOfBoundsException and occurs when we try to access illegal index for some method.

Ex:
1. Accessing value with charAt method with index which is not in range of that array.

```
public class StringIndexOutOfBounds {
	public static void main(String[] args) {
	String str = "Test";
	System.out.println(str.charAt(10));
	}
}
```

O/P:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out
of range: 10
        at java.lang.String.charAt(String.java:658)
        at StringIndexOutOfBounds.main(StringIndexOutOfBounds.java:4)
```

==

Test.java

```
public class Test {
        public static void main (String arr[]) {
                System.out.println("In main method of Test class");
        }
}
```

javac Test.java

Edit Test.class and remove last two character and run:
java Test

```
java Test
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.ClassFormatError: Truncated class file
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:495)
```

```
Unknown constant tag 108 in class file Test
Incompatible magic value 4273651200 in class file Test
Illegal UTF8 string in constant pool in class file Test
LineNumberTable attribute has wrong length in class file Test
```

==

NullPointerException occurs when to we try to perform some operation by considering it as java object.

Ex:

1. While executing some method on null
2. While accessing some member variable on null
3. Trying to get length of null

```
public class NullPointer {
	public String msg;
	public static void main(String[] args) {
	NullPointer obj = null;
	String str = null;
	try {
	System.out.println(obj.msg);
	} catch (Exception ex) {
	ex.printStackTrace();
	}
	try {
	System.out.println(str.length());
	} catch (Exception ex) {
	ex.printStackTrace();
	}
	System.out.println("In main method of NullPointer class");
	}
}
```

O/P:

```
java.lang.NullPointerException
	at NullPointer.main(Null.java:7)
java.lang.NullPointerException
	at NullPointer.main(Null.java:12)
In main method of NullPointer class
```

Basic Runtime constructor (2)

==

ArithmeticException occurs when we try to perform some arithmetic operation which is incorrect.

Ex:

1. While deviding zero by any number. (10/0)
2. While getting modulo of any number by zero. (10 % 0)

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

O/P:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Arithmetic.main(Arithmetic.java:4)
```

Basic Runtime constructor (2)

==

Do we always get ArithmeticException while performing operation devide by zero?

float or double

```
public static void main(String[] args) {
	float input = 0;
	System.out.println(10/input);
}
```

O/P:
Infinity

==

ArrayIndexOutOfBoundsException occurs when we try to perform some illegal index of an array.

Ex:

1. Accessing index which is not in range of that array.
2. Assigning value to index greater than size of that array.

```
public class ArrayIndexOutOfBounds {
	public static void main(String[] args) {
	String[] arr = new String[2];
	arr[0] = "Test1";
	arr[1] = "Test2";
	try {
	arr[2] = "Test3";
	} catch (Exception ex) {
	ex.printStackTrace();
	}
	try {
	System.out.println(arr[3]);
	} catch (Exception ex) {
	ex.printStackTrace();
	}
	}
}
```

O/P:

```
java.lang.ArrayIndexOutOfBoundsException: 2
	at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:7)
java.lang.ArrayIndexOutOfBoundsException: 3
	at ArrayIndexOutOfBounds.main(ArrayIndexOutOfBounds.java:12)
```

==

StringIndexOutOfBoundsException is also similar to ArrayIndexOutOfBoundsException and occurs when we try to access illegal index for some method.

Ex:

1. Accessing value with charAt method with index which is not in range of that array.

```
public class StringIndexOutOfBounds {
	public static void main(String[] args) {
	String str = "Test";
	System.out.println(str.charAt(10));
	}
}
```

O/P:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 10
	at java.lang.String.charAt(String.java:658)
	at StringIndexOutOfBounds.main(StringIndexOutOfBounds.java:4)
```

==

ClassCastException occurs when we try to type cast any object into any unrelated class. In below code we are passing String object in m1 method, in same m1 method we can type cast it to String only, type casting it to any other unrelated class will give ClassCastException.

```
public class ClassCast {
	public static void main(String[] args) {
	String s1 = "abc";
	ClassCast.m1(s1);
	}
	public static void m1(Object obj) {
	String s2 = (String)obj;
	System.out.println(s2);
	StringBuilder sb = (StringBuilder)obj;
	System.out.println(sb);
	}
}
```

O/P:

```
abc
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.StringBuilder
	at ClassCast.m1(ClassCast.java:9)
	at ClassCast.main(ClassCast.java:4)
```

We can type cast object of any class to any of its super class hierarchy or to interface.

Ex:

```
	public class ClassCast {
	public static void main(String[] args) {
	C c1 = new C();
	ClassCast.m1(c1);
	}
	public static void m1(Object obj) {
	A a1 = (A)obj;
	System.out.println("C class object type casted to A class successfully.");
	I3 i3 = (I3)obj;
	System.out.println("C class object type casted to I3 interface successfully.");
	I1 i1 = (I1)obj;
	System.out.println("C class object type casted to I1 interface successfully.");
	B b1 = (B)obj;
	System.out.println("C class object type casted to B class successfully.");
	}
}
class A implements I1 {}
class B implements I2 {}
interface I1 {}
interface I2 {}
interface I3 {}
class C extends A implements I3 {
}
```

O/P:

```
C class object type casted to A class successfully.
C class object type casted to I3 interface successfully.
C class object type casted to I1 interface successfully.
Exception in thread "main" java.lang.ClassCastException: C cannot be cast to B

	at ClassCast.m1(ClassCast.java:13)
	at ClassCast.main(ClassCast.java:4)
```

Here object of type C can be type casted to:

1. A class, since C class extends from A class and we can use parent class to hold reference of child class.
2. I3 interface, since C class implements I3 interface and we can use interface to hold reference of implemented class.
3. I1 interface, since C extends A and A implements I1.

==

ArrayStoreException occurs when we try to add wrong type of object into array.

```
public class ArrayStore {
	public static void main(String[] args) {
	Object[] arr = new String[3];
	arr[0] = "ABC";
	arr[1] = 100;
	}
```

```
}
```

O/P:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
	at ArrayStore.main(ArrayStore.java:5)
```

Since we created String array of 3 size, we can add only String, trying to add any other type will give ArrayStoreException.

==

EnumConstantNotPresent exception occurs only when to try to access enum contant which is not present at runtime if it is used using annotation only.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
public class EnumConstantNotPresent {
	public static void main(String[] args) {
	System.out.println(TestClass.class.getAnnotation(MyAnnotation.class).value());
	}
}

@MyAnnotation(MyEnum.BOOK)
class TestClass {

}

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
	MyEnum value();
}

public enum MyEnum {
	BOOK, CONFIRM, CANCEL
}
```

O/P:

```
BOOK
```

Now remove BOOK from MyEnum class, compile code again and run EnumConstantNotPresent class.

Ex:

```
public enum MyEnum {
	CONFIRM, CANCEL
}
```

O/P:

```
vjoshi@vjoshi-Latitude-3540:~/Data/code/java$ java EnumConstantNotPresent
Exception in thread "main" java.lang.EnumConstantNotPresentException: MyEnum.BOOK
        at
sun.reflect.annotation.EnumConstantNotPresentExceptionProxy.generateException(Enum
ConstantNotPresentExceptionProxy.java:46)
        at
sun.reflect.annotation.AnnotationInvocationHandler.invoke(AnnotationInvocationHandler.
java:84)
        at $Proxy1.value(Unknown Source)
        at EnumConstantNotPresent.main(EnumConstantNotPresent.java:7)
```

Without annotation we never get above exception.

Ex:

```
public class EnumConstantNotPresent {
        public static void main(String[] args) {
        System.out.println(MyEnum.BOOK);
        }
}

public enum MyEnum {
        BOOK, CONFIRM, CANCEL
}
```

O/P:

```
BOOK
```

Now remove BOOK from MyEnum class, compile code again and run EnumConstantNotPresent class.

O/P:

```
Exception in thread "main" java.lang.NoSuchFieldError: BOOK
        at EnumConstantNotPresent.main(EnumConstantNotPresent.java:3)
```

We will get NoSuchFieldError.

==

CloneNotSupportedException occurs when we try to clone object of any class which does not support cloning.

Ex:

```
public class CloneNotSupported {
        public static void main(String[] args) throws CloneNotSupportedException {
        CloneNotSupported c1 = new CloneNotSupported();
        CloneNotSupported c2 = (CloneNotSupported) c1.clone();
        }
}
```

O/P:

```
Exception in thread "main" java.lang.CloneNotSupportedException: CloneNotSupported
```

```
	at java.lang.Object.clone(Native Method)
	at CloneNotSupported.main(CloneNotSupported.java:4)
```

With clone method JVM follows below steps:

Check if that class or any of its super class hierarchy implemented Cloneable interface.
	-> If not implemented it directly throws CloneNotSupportedException
	-> If implemented it checks again if clone method overridden
		-> If overridden, call clone method of same class
		-> If not, call clone method of object or parent class hierarchy if any.

==

IllegalThreadStateException occurs when thread is not in appropriate state. This exception will be thrown for below cases:

1. If we try to start already running thread again.
2. If we try to make thread daemon once it is started or normal thread once it is initially set as daemon.

Ex for case 1:

```
public class IllegalThreadState {
	public static void main(String[] args) throws InterruptedException {
	Thread t = new Thread() {
	public void run() {
		System.out.println("In Thread class");
	}
	};
	t.start();
	t.start();
	}
}
```

O/P:

```
In Thread classException in thread "main"
java.lang.IllegalThreadStateException
	at java.lang.Thread.start(Thread.java:705)
	at IllegalThreadState.main(IllegalThreadState.java:9)
```

Ex for case 2:

```
public class IllegalThreadState {
	public static void main(String[] args) throws InterruptedException {
	Thread t = new Thread() {
	public void run() {
		System.out.println("In Thread class");
	}
	};
	t.start();
	t.setDaemon(true);
```

```
	}
}
```

O/P:

```
In Thread classException in thread "main"
java.lang.IllegalThreadStateException
	at java.lang.Thread.setDaemon(Thread.java:1352)
	at IllegalThreadState.main(IllegalThreadState.java:9)
```

==

NumberFormatException occurs when we try to convert incorrect String into number (int / float / double / long / short). It can occur for below cases:

1. When string provided is not in number format.
2. When string provided crosses the range of converted number type.

Ex for case 1:

```
public class NumberFormat {
	public static void main(String[] args) {
	String str1 = "100";
	String str2 = "Test";
	System.out.println(Integer.parseInt(str1));
	System.out.println(Integer.parseInt(str2));
	}
}
```

O/P:

```
100
Exception in thread "main" java.lang.NumberFormatException: For input string: "Test"
	at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
	at java.lang.Integer.parseInt(Integer.java:580)
	at java.lang.Integer.parseInt(Integer.java:615)
	at NumberFormat.main(NumberFormat.java:6)
```

Ex for case 2:

```
public class NumberFormat {
	public static void main(String[] args) {
	String str1 = "100000000000";
	System.out.println(Integer.parseInt(str1));
	}
}
```

O/P:

```
Exception in thread "main" java.lang.NumberFormatException: For input string:
"100000000000"
	at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
```

```
        at java.lang.Integer.parseInt(Integer.java:583)
        at java.lang.Integer.parseInt(Integer.java:615)
        at NumberFormat.main(NumberFormat.java:4)
```

==

IllegalMonitorStateException occurs when we try to call wait, notify or notifyAll method for any object from thread without owning lock of that object.

Ex:

```
public class IllegalMonitorState {
        public static void main(String[] args) throws InterruptedException {
        MyThread mt = new MyThread();
        mt.start();
        mt.wait();
        }
}

class MyThread extends Thread {
        @Override
        public void run() {
        while (true) {
        System.out.println("In MyThread class");
        try {
                sleep(2000);
        } catch (InterruptedException ex) {
                ex.printStackTrace();
        }
        }
        }
}
```

O/P:

```
Exception in thread "main" In MyThread class
java.lang.IllegalMonitorStateException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:502)
        at IllegalMonitorState.main(IllegalMonitorState.java:5)
In MyThread class
In MyThread class
...
```

==

```
Throwable (1.0)
        ->      Exception (1.0)
                ->      RuntimeException (1.0)
                        ->      ArithmeticException (1.0)
                        ->      IndexOutOfBoundsException (1.0)
                                ->      ArrayIndexOutOfBoundsException (1.0)
```

```
                        ->      StringIndexOutOfBoundsException (1.0)
                ->      ArrayStoreException (1.0)
                ->      ClassCastException (1.0)
                ->      EnumConstantNotPresentException (1.5)
                ->      IllegalArgumentException (1.0)
                        ->      IllegalThreadStateException (1.0)
                        ->      NumberFormatException (1.0)
                ->      IllegalMonitorStateException (1.0)
                ->      IllegalStateException (1.1)
                ->      NullPointerException (1.0)
                ->      SecurityException (1.0)
                ->      TypeNotPresentException (1.5)
                ->      UnsupportedOperationException (1.2)
        ->      ReflectiveOperationException (1.7)
                ->      ClassNotFoundException (1.0)
                ->      IllegalAccessException (1.0)
                ->      InstantiationException (1.0)
                ->      NoSuchFieldException (1.1)
        ->      CloneNotSupportedException (1.0)
        ->      InterruptedException (1.0)
->      Error (1.0)
        ->      ThreadDeath (1.0)
        ->      AssertionError (1.4)
        ->      LinkageError (1.0)
                ->      BootstrapMethodError (1.7)
                ->      ClassCircularityError (1.0)
                ->      ClassFormatError (1.0)
                        ->      UnsupportedClassVersionError (1.2)
                ->      ExceptionInInitializerError (1.1)
                ->      IncompatibleClassChangeError (1.0)
                        ->      IllegalAccessError (1.0)
                        ->      NoSuchFieldError (1.0)
                        ->      InstantiationError (1.0)
                        ->      NoSuchMethodError (1.0)
                ->      NoClassDefFoundError (1.0)
                ->      UnsatisfiedLinkError (1.0)
                ->      VerifyError (1.0)
        ->      VirtualMachineError (1.0)
                ->      InternalError (1.0)
                ->      OutOfMemoryError (1.0)
                ->      StackOverflowError (1.0)
                ->      UnknownError (1.0)
```

the general purpose exception-chaining mechanism.

objects may be constructed by the virtual
 * machine as if {@linkplain Throwable#Throwable(String, Throwable,
 * boolean, boolean) suppression were disabled and/or the stack trace was not
 * writable}.

public final class StackTraceElement implements java.io.Serializable {

java.lang.SuppressWarnings

========

package java.lang;
Throwable implements Serializable

The {@code Throwable} class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this
 * class (or one of its subclasses) are thrown by the Java Virtual Machine or
 * can be thrown by the Java {@code throw} statement. Similarly, only
 * this class or one of its subclasses can be the argument type in a
 * {@code catch} clause.

 * For the purposes of compile-time checking of exceptions, {@code
 * Throwable} and any subclass of {@code Throwable} that is not also a
 * subclass of either {@link RuntimeException} or {@link Error} are
 * regarded as checked exceptions.

 * <p>Instances of two subclasses, {@link java.lang.Error} and
 * {@link java.lang.Exception}, are conventionally used to indicate
 * that exceptional situations have occurred. Typically, these instances
 * are freshly created in the context of the exceptional situation so
 * as to include relevant information (such as stack trace data).

  * <p>A throwable contains a snapshot of the execution stack of its
 * thread at the time it was created. It can also contain a message
 * string that gives more information about the error. Over time, a
 * throwable can {@linkplain Throwable#addSuppressed suppress} other
 * throwables from being propagated. Finally, the throwable can also
 * contain a cause: another throwable that caused this
 * throwable to be constructed. The recording of this causal information
 * is referred to as the chained exception facility, as the
 * cause can, itself, have a cause, and so on, leading to a "chain" of
 * exceptions, each caused by another.

 It would be bad
 * design to let the throwable thrown by the lower layer propagate outward, as
 * it is generally unrelated to the abstraction provided by the upper layer.

  * <p>A cause can be associated with a throwable in two ways: via a
 * constructor that takes the cause as an argument, or via the
 * {@link #initCause(Throwable)} method.

  * Because the {@code initCause} method is public, it allows a cause to be
 * associated with any throwable, even a "legacy throwable" whose
 * implementation predates the addition of the exception chaining mechanism to
 * {@code Throwable}.

```
 * <p>By convention, class {@code Throwable} and its subclasses have two
* constructors, one that takes no arguments and one that takes a
* {@code String} argument that can be used to produce a detail message.
* Further, those subclasses that might likely have a cause associated with
* them should have two more constructors, one that takes a
* {@code Throwable} (the cause), and one that takes a
* {@code String} (the detail message) and a {@code Throwable} (the
* cause).

 /** use serialVersionUID from JDK 1.0.2 for interoperability */
        private static final long serialVersionUID = -3042686055658047285L;

        /**
        * Native code saves some indication of the stack backtrace in this slot.
        */
        private transient Object backtrace;

        /**
        * Specific details about the Throwable.  For example, for
        * {@code FileNotFoundException}, this contains the name of
        * the file that could not be found.
        *
        * @serial
        */
        private String detailMessage;
```

==

```
public class Exception extends Throwable {

 * The class {@code Exception} and its subclasses are a form of
 * {@code Throwable} that indicates conditions that a reasonable
 * application might want to catch.

 * <p>The class {@code Exception} and any subclasses that are not also
 * subclasses of {@link RuntimeException} are <em>checked
 * exceptions</em>.  Checked exceptions need to be declared in a
 * method or constructor's {@code throws} clause if they can be thrown
 * by the execution of the method or constructor and propagate outside
 * the method or constructor boundary.

public Exception() { .. }
public Exception(String message) { .. }
public Exception(String message, Throwable cause) { .. }
public Exception(Throwable cause) { .. }
protected Exception(String message, Throwable cause, boolean enableSuppression,
boolean writableStackTrace) { .. }
```

There is no code written to perform any activity in any of the constructor, all constructor calls parent class constructor only.

Ex:
```
public Exception(String message) {
        super(message);
}
```

==

```
public class RuntimeException extends Exception {

 * {@code RuntimeException} is the superclass of those
 * exceptions that can be thrown during the normal operation of the
 * Java Virtual Machine.

  * <p>{@code RuntimeException} and its subclasses are <em>unchecked
 * exceptions</em>.  Unchecked exceptions do <em>not</em> need to be
 * declared in a method or constructor's {@code throws} clause if they
 * can be thrown by the execution of the method or constructor and
 * propagate outside the method or constructor boundary.
```

RuntimeException also contains 5 constructors which are written exactly same as Exception class and all constructor calls parent class constructor only.

Ex:
```
public RuntimeException(String message) {
        super(message);
}
```

==

```
public class Error extends Throwable {

 * An {@code Error} is a subclass of {@code Throwable}
 * that indicates serious problems that a reasonable application
 * should not try to catch. Most such errors are abnormal conditions.
 * The {@code ThreadDeath} error, though a "normal" condition,
 * is also a subclass of {@code Error} because most applications
 * should not try to catch it.

  * A method is not required to declare in its {@code throws}
 * clause any subclasses of {@code Error} that might be thrown
 * during the execution of the method but not caught, since these
 * errors are abnormal conditions that should never occur.

  * That is, {@code Error} and its subclasses are regarded as unchecked
 * exceptions for the purposes of compile-time checking of exceptions.
```

Error all 5 contructor same as Exceptions.

==

public class ThreadDeath extends Error {

 * An instance of {@code ThreadDeath} is thrown in the victim thread
 * when the (deprecated) {@link Thread#stop()} method is invoked.

 * <p>An application should catch instances of this class only if it
 * must clean up after being terminated asynchronously. If
 * {@code ThreadDeath} is caught by a method, it is important that it
 * be rethrown so that the thread actually dies.

==

public class ArithmeticException extends RuntimeException {

 * Thrown when an exceptional arithmetic condition has occurred. For
 * example, an integer "divide by zero" throws an
 * instance of this class.

 Only 2 constructor:

```
public ArithmeticException() {
        super();
}

public ArithmeticException(String s) {
        super(s);
}
```

==

public class IndexOutOfBoundsException extends RuntimeException {

 * Thrown to indicate that an index of some sort (such as to an array, to a
 * string, or to a vector) is out of range.
 * <p>
 * Applications can subclass this class to indicate similar exceptions.

Only 2 constructor:

```
public IndexOutOfBoundsException() {
        super();
}

public IndexOutOfBoundsException(String s) {
        super(s);
}
```

==

public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException {

```
 * Thrown to indicate that an array has been accessed with an
 * illegal index. The index is either negative or greater than or
 * equal to the size of the array.

3 constructors:

public ArrayIndexOutOfBoundsException() {
        super();
}

public ArrayIndexOutOfBoundsException(int index) {
        super("Array index out of range: " + index);
}

public ArrayIndexOutOfBoundsException(String s) {
        super(s);
}

==

public class ArrayStoreException extends RuntimeException {

 * Thrown to indicate that an attempt has been made to store the
 * wrong type of object into an array of objects. For example, the
 * following code generates an <code>ArrayStoreException</code>:
 * <blockquote><pre>
 *       Object x[] = new String[3];
 *       x[0] = new Integer(0);
 * </pre></blockquote>

public ArrayStoreException() {
        super();
}

public ArrayStoreException(String s) {
        super(s);
}

==

public class AssertionError extends Error {

10 contructor, 7 public one-argument constructor.

public AssertionError() {

}

private AssertionError(String detailMessage) {
        super(detailMessage);
}
```

```
public AssertionError(Object detailMessage) { .. }

public AssertionError(boolean detailMessage) {
        this(String.valueOf(detailMessage));
        }

public AssertionError(char detailMessage) {
        this(String.valueOf(detailMessage));
        }

public AssertionError(int detailMessage) {
        this(String.valueOf(detailMessage));
        }

public AssertionError(long detailMessage) {
        this(String.valueOf(detailMessage));
        }

public AssertionError(float detailMessage) {
        this(String.valueOf(detailMessage));
        }

public AssertionError(double detailMessage) {
        this(String.valueOf(detailMessage));
        }

public AssertionError(String message, Throwable cause) {
        super(message, cause);
        }
```

==

```
public
class LinkageError extends Error {

 * Subclasses of {@code LinkageError} indicate that a class has
 * some dependency on another class; however, the latter class has
 * incompatibly changed after the compilation of the former class.

public LinkageError() {
        super();
        }

public LinkageError(String s) {
        super(s);
        }

public LinkageError(String s, Throwable cause) {
        super(s, cause);
        }
```

==

public class BootstrapMethodError extends LinkageError {

 * Thrown to indicate that an {@code invokedynamic} instruction has
 * failed to find its bootstrap method,
 * or the bootstrap method has failed to provide a
 * {@linkplain java.lang.invoke.CallSite call site} with a {@linkplain
java.lang.invoke.CallSite#getTarget target}
 * of the correct {@linkplain java.lang.invoke.MethodHandle#type method type}.

 * @since 1.7

==

public class ClassCastException extends RuntimeException {

 * Thrown to indicate that the code has attempted to cast an object
 * to a subclass of which it is not an instance. For example, the
 * following code generates a <code>ClassCastException</code>:
 * <blockquote><pre>
 *   Object x = new Integer(0);
 *   System.out.println((String)x);
 * </pre></blockquote>

public ClassCastException() {
        super();
        }

public ClassCastException(String s) {
        super(s);
        }

==

public class ClassCircularityError extends LinkageError {

 * Thrown when the Java Virtual Machine detects a circularity in the
 * superclass hierarchy of a class being loaded.

public ClassCircularityError() {
        super();
        }

public ClassCircularityError(String s) {
        super(s);
        }

==

```
public class ClassFormatError extends LinkageError {

 * Thrown when the Java Virtual Machine attempts to read a class
 * file and determines that the file is malformed or otherwise cannot
 * be interpreted as a class file.

public ClassFormatError() {
        super();
        }

public ClassFormatError(String s) {
        super(s);
        }

==

public class ReflectiveOperationException extends Exception {

 * @since 1.7

 * Common superclass of exceptions thrown by reflective operations in
 * core reflection.

public ReflectiveOperationException() {
        super();
        }

public ReflectiveOperationException(String message) {
        super(message);
        }

public ReflectiveOperationException(String message, Throwable cause) {
        super(message, cause);
        }

public ReflectiveOperationException(Throwable cause) {
        super(cause);
        }

==

public class ClassNotFoundException extends ReflectiveOperationException {

 * Thrown when an application tries to load in a class through its
 * string name using:
 * <ul>
 * <li>The <code>forName</code> method in class <code>Class</code>.
 * <li>The <code>findSystemClass</code> method in class
 *      <code>ClassLoader</code> .
 * <li>The <code>loadClass</code> method in class <code>ClassLoader</code>.
 * </ul>
```

```
 * <p>
 * but no definition for the class with the specified name could be found.

public ClassNotFoundException() {
        super((Throwable)null);  // Disallow initCause
        }

public ClassNotFoundException(String s) {
        super(s, null);  //  Disallow initCause
        }

public ClassNotFoundException(String s, Throwable ex) {
        super(s, null);  //  Disallow initCause
        this.ex = ex;
        }

public Throwable getException() {
        return ex;
        }

public Throwable getCause() {
        return ex;
        }

==

public class CloneNotSupportedException extends Exception {

 * Thrown to indicate that the <code>clone</code> method in class
 * <code>Object</code> has been called to clone an object, but that
 * the object's class does not implement the <code>Cloneable</code>
 * interface.

public CloneNotSupportedException() {
        super();
        }

public CloneNotSupportedException(String s) {
        super(s);
        }

==

public class EnumConstantNotPresentException extends RuntimeException {

 * Thrown when an application tries to access an enum constant by name
 * and the enum type contains no constant with the specified name.
 * This exception can be thrown by the {@linkplain
 * java.lang.reflect.AnnotatedElement API used to read annotations
 * reflectively}.
```

```
 * @since   1.5

 public EnumConstantNotPresentException(Class<? extends Enum> enumType,
                                        String constantName) {
        super(enumType.getName() + "." + constantName);
        this.enumType = enumType;
        this.constantName  = constantName;
        }

==

public class ExceptionInInitializerError extends LinkageError {

 * Signals that an unexpected exception has occurred in a static initializer.
 * An <code>ExceptionInInitializerError</code> is thrown to indicate that an
 * exception occurred during evaluation of a static initializer or the
 * initializer for a static variable.

public ExceptionInInitializerError() {
        initCause(null);  // Disallow subsequent initCause
        }

public ExceptionInInitializerError(Throwable thrown) {
        initCause(null);  // Disallow subsequent initCause
        this.exception = thrown;
        }

public ExceptionInInitializerError(String s) {
        super(s);
        initCause(null);  // Disallow subsequent initCause
        }

==

public class IncompatibleClassChangeError extends LinkageError {

 * Thrown when an incompatible class change has occurred to some class
 * definition. The definition of some class, on which the currently
 * executing method depends, has since changed.

public IncompatibleClassChangeError () {
        super();
        }

        public IncompatibleClassChangeError(String s) {
        super(s);
        }

==

public class IllegalAccessError extends IncompatibleClassChangeError {
```

```
 * Thrown if an application attempts to access or modify a field, or
 * to call a method that it does not have access to.

public IllegalAccessError() {
        super();
        }

        public IllegalAccessError(String s) {
        super(s);
        }

==

public class IllegalAccessException extends ReflectiveOperationException {

 * An IllegalAccessException is thrown when an application tries
 * to reflectively create an instance (other than an array),
 * set or get a field, or invoke a method, but the currently
 * executing method does not have access to the definition of
 * the specified class, field, method or constructor.

public IllegalAccessException() {
        super();
        }

public IllegalAccessException(String s) {
        super(s);
        }

==

public class IllegalArgumentException extends RuntimeException {

 * Thrown to indicate that a method has been passed an illegal or
 * inappropriate argument.

public IllegalArgumentException() {
        super();
        }

        public IllegalArgumentException(String s) {
        super(s);
        }

        public IllegalArgumentException(String message, Throwable cause) {
        super(message, cause);
        }

        public IllegalArgumentException(Throwable cause) {
        super(cause);
```

```
        }

==

public class IllegalMonitorStateException extends RuntimeException {

 * Thrown to indicate that a thread has attempted to wait on an
 * object's monitor or to notify other threads waiting on an object's
 * monitor without owning the specified monitor.

public IllegalMonitorStateException() {
        super();
        }

public IllegalMonitorStateException(String s) {
        super(s);
        }

==

public class IllegalStateException extends RuntimeException {

 * Signals that a method has been invoked at an illegal or
 * inappropriate time.  In other words, the Java environment or
 * Java application is not in an appropriate state for the requested
 * operation.


public IllegalStateException() {
        super();
        }

        public IllegalStateException(String s) {
        super(s);
        }

        public IllegalStateException(String message, Throwable cause) {
        super(message, cause);
        }

        public IllegalStateException(Throwable cause) {
        super(cause);
        }

==

public class IllegalThreadStateException extends IllegalArgumentException {

 * Thrown to indicate that a thread is not in an appropriate state
 * for the requested operation. See, for example, the
 * <code>suspend</code> and <code>resume</code> methods in class
```

```
 * <code>Thread</code>.

public IllegalThreadStateException() {
        super();
        }

        public IllegalThreadStateException(String s) {
        super(s);
        }

==

public class IncompatibleClassChangeError extends LinkageError {

 * Thrown when an incompatible class change has occurred to some class
 * definition. The definition of some class, on which the currently
 * executing method depends, has since changed.


public IncompatibleClassChangeError () {
        super();
        }

public IncompatibleClassChangeError(String s) {
        super(s);
        }

==

public class InstantiationError extends IncompatibleClassChangeError {

 * Thrown when an application tries to use the Java <code>new</code>
 * construct to instantiate an abstract class or an interface.

public InstantiationError() {
        super();
        }

        public InstantiationError(String s) {
        super(s);
        }

==

public class InstantiationException extends ReflectiveOperationException {

 * Thrown when an application tries to create an instance of a class
 * using the {@code newInstance} method in class
 * {@code Class}, but the specified class object cannot be
 * instantiated.  The instantiation can fail for a variety of
 * reasons including but not limited to:
```

```
 *
 * <ul>
 * <li> the class object represents an abstract class, an interface,
 *      an array class, a primitive type, or {@code void}
 * <li> the class has no nullary constructor
 *</ul>

public InstantiationException() {
        super();
        }

        public InstantiationException(String s) {
        super(s);
        }

==

abstract public class VirtualMachineError extends Error {

 * Thrown to indicate that the Java Virtual Machine is broken or has
 * run out of resources necessary for it to continue operating.

public VirtualMachineError() {
        super();
        }

        public VirtualMachineError(String message) {
        super(message);
        }

        public VirtualMachineError(String message, Throwable cause) {
        super(message, cause);
        }

        public VirtualMachineError(Throwable cause) {
        super(cause);
        }

==

public class InternalError extends VirtualMachineError {

 * Thrown to indicate some unexpected internal error has occurred in
 * the Java Virtual Machine.

public InternalError() {
        super();
        }

        public InternalError(String message) {
        super(message);
```

```
        }

        public InternalError(String message, Throwable cause) {
        super(message, cause);
        }

        public InternalError(Throwable cause) {
        super(cause);
        }

==

public class InterruptedException extends Exception {

 * Thrown when a thread is waiting, sleeping, or otherwise occupied,
 * and the thread is interrupted, either before or during the activity.
 * Occasionally a method may wish to test whether the current
 * thread has been interrupted, and if so, to immediately throw
 * this exception.  The following code can be used to achieve
 * this effect:
 * <pre>
 *  if (Thread.interrupted())  // Clears interrupted status!
 *      throw new InterruptedException();
 * </pre>

public InterruptedException() {
        super();
        }

        public InterruptedException(String s) {
        super(s);
        }

==

public class NegativeArraySizeException extends RuntimeException {

 * Thrown if an application tries to create an array with negative size.

public NegativeArraySizeException() {
        super();
        }

        public NegativeArraySizeException(String s) {
        super(s);
        }

==

public class NoClassDefFoundError extends LinkageError {
```

```
 * Thrown if the Java Virtual Machine or a <code>ClassLoader</code> instance
 * tries to load in the definition of a class (as part of a normal method call
 * or as part of creating a new instance using the <code>new</code> expression)
 * and no definition of the class could be found.

public NoClassDefFoundError() {
        super();
        }

        public NoClassDefFoundError(String s) {
        super(s);
        }

==

public class NoSuchFieldError extends IncompatibleClassChangeError {

 * Thrown if an application tries to access or modify a specified
 * field of an object, and that object no longer has that field.

public NoSuchFieldError() {
        super();
        }

        public NoSuchFieldError(String s) {
        super(s);
        }

==

public class NoSuchFieldException extends ReflectiveOperationException {

 * Signals that the class doesn't have a field of a specified name.

public NoSuchFieldException() {
        super();
        }

        public NoSuchFieldException(String s) {
        super(s);
        }

==

public class NoSuchMethodError extends IncompatibleClassChangeError {

 * Thrown if an application tries to call a specified method of a
 * class (either static or instance), and that class no longer has a
 * definition of that method.

public NoSuchMethodError() {
```

```
        super();
        }

        public NoSuchMethodError(String s) {
        super(s);
        }

==

public class NoSuchMethodException extends ReflectiveOperationException {

 * Thrown when a particular method cannot be found.

public NoSuchMethodException() {
        super();
        }

        public NoSuchMethodException(String s) {
        super(s);
        }

==

public class NullPointerException extends RuntimeException {

 * Thrown when an application attempts to use {@code null} in a
 * case where an object is required. These include:

 * Applications should throw instances of this class to indicate
 * other illegal uses of the {@code null} object.

public NullPointerException() {
        super();
        }

        public NullPointerException(String s) {
        super(s);
        }

==

public class NumberFormatException extends IllegalArgumentException {

 * Thrown to indicate that the application has attempted to convert
 * a string to one of the numeric types, but that the string does not
 * have the appropriate format.

public NumberFormatException () {
        super();
        }
```

```
        public NumberFormatException (String s) {
        super (s);
        }

==

public class OutOfMemoryError extends VirtualMachineError {

 * Thrown when the Java Virtual Machine cannot allocate an object
 * because it is out of memory, and no more memory could be made
 * available by the garbage collector.

public OutOfMemoryError() {
        super();
        }

        public OutOfMemoryError(String s) {
        super(s);
        }

==

public class SecurityException extends RuntimeException {

 * Thrown by the security manager to indicate a security violation.

public SecurityException() {
        super();
        }

        public SecurityException(String s) {
        super(s);
        }

        public SecurityException(String message, Throwable cause) {
        super(message, cause);
        }

        public SecurityException(Throwable cause) {
        super(cause);
        }

==

public class StackOverflowError extends VirtualMachineError {

 * Thrown when a stack overflow occurs because an application

public StackOverflowError() {
        super();
        }
```

```
        public StackOverflowError(String s) {
        super(s);
        }

==

public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException {


 * Thrown by {@code String} methods to indicate that an index
 * is either negative or greater than the size of the string.  For
 * some methods such as the charAt method, this exception also is
 * thrown when the index is equal to the size of the string.

public StringIndexOutOfBoundsException() {
        super();
        }

        public StringIndexOutOfBoundsException(String s) {
        super(s);
        }

        public StringIndexOutOfBoundsException(int index) {
        super("String index out of range: " + index);
        }

==

public class TypeNotPresentException extends RuntimeException {

 * Thrown when an application tries to access a type using a string
 * representing the type's name, but no definition for the type with
 * the specified name can be found.   This exception differs from
 * {@link ClassNotFoundException} in that <tt>ClassNotFoundException</tt> is a
 * checked exception, whereas this exception is unchecked.

public TypeNotPresentException(String typeName, Throwable cause) {
        super("Type " + typeName + " not present", cause);
        this.typeName = typeName;
        }

==

public class UnknownError extends VirtualMachineError {

 * Thrown when an unknown but serious exception has occurred in the
 * Java Virtual Machine.

public UnknownError() {
        super();
```

```
        }

        public UnknownError(String s) {
        super(s);
        }

==

public class UnsatisfiedLinkError extends LinkageError {

 * Thrown if the Java Virtual Machine cannot find an appropriate
 * native-language definition of a method declared <code>native</code>.

public UnsatisfiedLinkError() {
        super();
        }

        public UnsatisfiedLinkError(String s) {
        super(s);
        }

==

public class UnsupportedClassVersionError extends ClassFormatError {

 * Thrown when the Java Virtual Machine attempts to read a class
 * file and determines that the major and minor version numbers
 * in the file are not supported.

public UnsupportedClassVersionError() {
        super();
        }

        public UnsupportedClassVersionError(String s) {
        super(s);
        }

==

public class UnsupportedOperationException extends RuntimeException {

 * Thrown to indicate that the requested operation is not supported.<p>

public UnsupportedOperationException() {
        }

        public UnsupportedOperationException(String message) {
        super(message);
        }

        public UnsupportedOperationException(String message, Throwable cause) {
```

```
        super(message, cause);
        }

        public UnsupportedOperationException(Throwable cause) {
        super(cause);
        }
```

==

```
public class VerifyError extends LinkageError {

 * Thrown when the "verifier" detects that a class file,
 * though well formed, contains some sort of internal inconsistency
 * or security problem.

public VerifyError() {
        super();
        }

        public VerifyError(String s) {
        super(s);
        }
```

==

What will happen if we try to compile file which does not exists? Does compiler handle it as exception or error?

We will get "file not found" message which is initial step before checking exception and error. Exceptions and errors can occur only if file exists.

```
javac test.java (If test.java does not exist):
javac: file not found: test.java
Usage: javac <options> <source files>
use -help for a list of possible options

javac test.txt (If test.txt does not exist):
javac: invalid flag: test.txt
Usage: javac <options> <source files>
use -help for a list of possible options
```

==

Create new file with name NewTest.java:

```
public class Test {
        public static void main (String arr[]) {
                System.out.println("In main method of Test class");
        }
}
```

javac NewTest.java

O/P:
```
NewTest.java:1: error: class Test is public, should be declared in a file named Test.java
public class Test {
       ^
1 error
```

==

ExceptionInInitializerError occurs if some error occured while initializing static member variable or while executing static block.

Ex 1:
ExceptionInInitializer.java
```
public class ExceptionInInitializer {
	static Object obj = new Test();
	public static void main(String[] args) {
	System.out.println("In main method");
	}
}
```

Test.java
```
public class Test {
}
```

Compile both classes, delete Test.class and run ExceptionInInitializer class.

O/P:
```
Exception in thread "main" java.lang.NoClassDefFoundError: Test
	at ExceptionInInitializer.<clinit>(ExceptionInInitializer.java:2)
Caused by: java.lang.ClassNotFoundException: Test
	at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
	at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
	at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
	at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
	... 1 more
```

Ex 2:
```
public class ExceptionInInitializer {
	static int i = 1 / 0;
	public static void main(String[] args) {
	System.out.println("In main method");
	}
}
```

O/P:
```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArithmeticException: / by zero
	at ExceptionInInitializer.<clinit>(ExceptionInInitializer.java:2)
```

Ex 3:

```
public class ExceptionInInitializer {
	static {
	int i = 1 / 0;
	}
	public static void main(String[] args) {
	System.out.println("In main method");
	}
}
```

O/P:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArithmeticException: / by zero
	at ExceptionInInitializer.<clinit>(ExceptionInInitializer.java:3)
```

==

IllegalStateException occurs when an application tries to call any method at an illegal or inappropriate time.

Ex:

```
import java.util.HashSet;
import java.util.Set;
public class IllegalState {
	public static void main(String[] args) {
	Set<String> set = new HashSet<>();
	set.add("Test1");
	set.add("Test2");
	set.iterator().remove();
	}
}
```

O/P:

```
Exception in thread "main" java.lang.IllegalStateException
	at java.util.HashMap$HashIterator.remove(HashMap.java:1449)
	at IllegalState.main(IllegalState.java:8)
```

Here we are calling remove method directly with iterator even without iterating it.

==

SecurityException occurs when an application tries to violate any security constraint. It can occue for below cases:
1. If an application tries to use same package name which is already reserved by Java. (Ex: java, java.lang, java.util)

Ex for Case 1:

```
package java;
public class Security {
	public static void main(String[] args) {
	System.out.println("In main");
```

```
        }
}
```

Compile code: javac java/Security.java
Run class file: java java.Security

O/P:

```
vjoshi@vjoshi-Latitude-3540:~/Data/code/java$ java java.Security
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.SecurityException: Prohibited package name: java
        at java.lang.ClassLoader.preDefineClass(ClassLoader.java:662)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:761)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:495)
```

==

What will happen if two classes are declared as subclass of each other.

```
public class A extends B {

}
class B extends A {

}
```

In this case compiler will throw compilation error.
O/P:

```
A.java:1: error: cyclic inheritance involving A
public class A extends B {
       ^
1 error
```

==

UnsupportedOperationException occurs when an application tries to perform some operation which is not supported. It can occure for below cases:

1. When we call add/remove method on unmodifiable collection object.
2. If we implement any interface method or extend abstract class method which is not supported by that class. If we implement interface or extend abstract class without implemeting required method all IDEs provide default option to throw this exception.

Code ex 1:

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
public class UnsupportedOperation {
        public static void main(String[] args) {
        List<String> list = new LinkedList<>();
        list.add("Test 1");
        list.add("Test 2");
        list = Collections.unmodifiableList(list);
        System.out.println(list);
        list.add("Test 3");
        }
}
```

O/P:

```
[Test 1, Test 2]
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
        at UnsupportedOperation.main(UnsupportedOperation.java:11)
```

Code Ex 2:

```
public class UnsupportedOperation implements I1 {
        public static void main(String[] args) {
        System.out.println("In Main");
        UnsupportedOperation uo = new UnsupportedOperation();
        uo.m1();
        }
        @Override
        public void m1() {
        throw new UnsupportedOperationException("Not supported yet.");
        }
}
interface I1 {
        public void m1();
}
```

O/P:

```
In Main
Exception in thread "main" java.lang.UnsupportedOperationException: Not supported
yet.
   at ex.UnsupportedOperation.m1(UnsupportedOperation.java:9)
   at ex.UnsupportedOperation.main(UnsupportedOperation.java:5)
/home/vjoshi/.cache/netbeans/8.1/executor-snippets/run.xml:53: Java returned: 1
```

==

END