Marker Interface

It is the specical kind of interface which is empty ie. It does not have any data field and abstract method. It is used to give information to the compiler to do some special kind of task.

```
 interface I
{
//nothing
}
```

Example: **Serializable, Cloneable**

```
class student implements Serializable,Cloneable
{
 int roll;
  string name;
  student(int roll,String s)
  { this.roll=roll;
     name=s;
}
}
student s1=new student(1,"bbb");
student s2=new student(2,"bbb");
 student s3=(student)s1.clone();
```

Exception Handling

There are two types of errors in a program.

* **Syntax errors:** Also called compile time errors when a program violates the rules of the programming language. Ex: missing a semi colon, typing a spelling mistake in keyword etc.
* **Run-time errors:** Errors which are raised during execution of the program due to violation of semantic rules or other abnormal behavior. Ex: dividing a number with zero, stack overflow, illegal type conversion, accessing an unavailable array element etc.

* **Logic errors:** Errors which are raised during the execution of the program due to mistakes in the logic applied in the program. These are the most severe kind of errors to detect. Ex: misplacing a minus sign in the place of plus sign, placing a semi colon at the end of loops etc.

Exception:

* An abnormal condition that occurs at run-time
* A run-time error is known as an exception.

* Exceptions are predefined objects in java.

Exception Handling:

* Handling exceptions is known as exception handling.
* Instead of letting the program terminate abnormally when an error occurs at run-time.we provide meaningful error messages to the users by handling the exception(s).

Hierarchy of Java Exception classes , which are present in java.lang package.

Exceptions are again categorized into **checked** and **unchecked** exception.

Checked Exception

Example

Unchecked Exception

ArithmeticException

Java provide five keywords for exception handling

***try, catch, throw, throws,* and *finally*.**

* **try:** Statements that are supposed to raise exception(s) are placed inside a *try* block.
* **catch:** Code that handles exceptions are placed in *catch* blocks. A *try* block must be followed by one or more *catch* blocks or a single *finally* block.
* **throw:** Most of the exceptions are thrown automatically by the Java run-time system. We can throw exceptions manually using *throw* keyword.
* **throws:** If a method which raises exceptions doesn't want to handle them, they can be thrown

to parent method or the Java run-time using the *throws* keyword.

* **finally:** All the statements that should execute irrespective of whether a exception arises or not is placed in the *finally* block.

try block

* Java try block is used to enclose the code that might throw an exception.
* It must be used within the method.
* Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
try{
//code that may throw exception
}
catch(Exception_class_Name reference){}
```

Syntax of try-finally block

```
try{
//code that may throw exception
}finally{}
```

Java catch block

* Java catch block is used to handle the Exception. It must be used after the try block only.
* multiple catch block can be used with a single try.

Problem without exception handling

```
public class Testtrycatch1{
public static void main(String args[]){

int data=50/0;//may throw exception
System.out.println("rest of the code...");
}
 }
```

Output:

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

By exception handling

```
public class Testtrycatch2{
public static void main(String args[])
{ try{
int data=50/0;
}

catch(ArithmeticException e)

{

System.out.println(e);

}

System.out.println("rest of the code...");
} }
```

1.Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero rest of the code...
```

Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

```
public class TestMultipleCatchBlock{

public static void main(String args[])
{
try{
int a[]=new int[5];
a[5]=30/0;
}
catch(ArithmeticException e)
{
System.out.println("task1 is completed");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("task 2 completed");
}
catch(Exception e)
{
System.out.println("common task completed");
 }
System.out.println("rest of the code...");
} }
```

Output:

```
task1 completed rest of the code...
```

Note:

When multiple catch blocks are there for a try block, then the order of catch blocks must be from specific exception type to generalized exception type.

If there are ArithmeticException and Excetion type in catch blocks ,the catch block which has ArithmeticException argument must be written first .

Because the the super class reference such as Exception reference can refer to sub class object like ArithmeticException.

Nested try example

```
class Excep6{
public static void main(String args[]){
try{
try{
System.out.println("going to divide");
int b =39/0;
}catch(ArithmeticException e)

{System.out.println(e);}
try{
int a[]=new int[5];
a[5]=4;
}catch(ArrayIndexOutOfBoundsException e)
{System.out.println(e);}
System.out.println("other statement);
}catch(Exception e)
{System.out.println("handeled");}
System.out.println("normal flow..");
}
}
```

finally block

* **It** is a block that is used *to execute important code* such as closing connection, stream etc.
* Java finally block is always executed whether exception is handled or not.
* Java finally block follows try or catch block.

Case 1

finally example where **exception doesn't occur**.

```
classTestFinallyBlock{
public static void main(String args[])
{
try
{
int data=25/5;
System.out.println(data);
}
catch(NullPointerException e)
{
System.out.println(e);
}
finally
{
System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
}
```

```
Output:5 finally block is always executed rest of the code...
```

Java throw keyword

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

throw exceptionobject;

Java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{

static void validate(int age)
{ if(age<18)
throw new ArithmeticException("not valid");
 else
System.out.println("welcome to vote");
}
public static void main(String args[])
{
 validate(13);
System.out.println("rest of the code...");
 } }
```

Output:

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

Java throws keyword

- The **Java throws keyword** is used to declare an exception. It gives information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions. The exceptions are handled by the caller of the method where the exceptions arised.

Syntax of java throws

```
return_type method_name() throws exception_class_name
{
//method code
}
```

Java throws example

java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
void m() throws IOException
{
throw new IOException("device error");//checked exception
}
void n() throws IOException
{ m();
}
Void p(){
try{
n();
}
catch(Exception e){System.out.println("exception handled");}
}
public static void main(String args[])
{
Testthrows1 obj=new Testthrows1();
obj.p();
System.out.println("normal flow...");
 }
 }
```

```
 Output:
exception handled

normal flow...
```

Java Custom Exception

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

Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

```
class InvalidAgeException extends Exception{
InvalidAgeException(String s)
{
super(s);
}
 }
class TestCustomException1
{
static void validate(int age) throws InvalidAgeException
{ if(age<18)
throw new InvalidAgeException("not valid");
else
System.out.println("welcome to vote");
}
public static void main(String args[])
{ try
{
```

```
validate(13);
}
catch(Exception m)
{System.out.println("Exception occured: "+m);}
System.out.println("rest of the code...");
 }
 }
```

```
Output:Exceptionoccured: InvalidAgeException:not valid rest of the code..
```