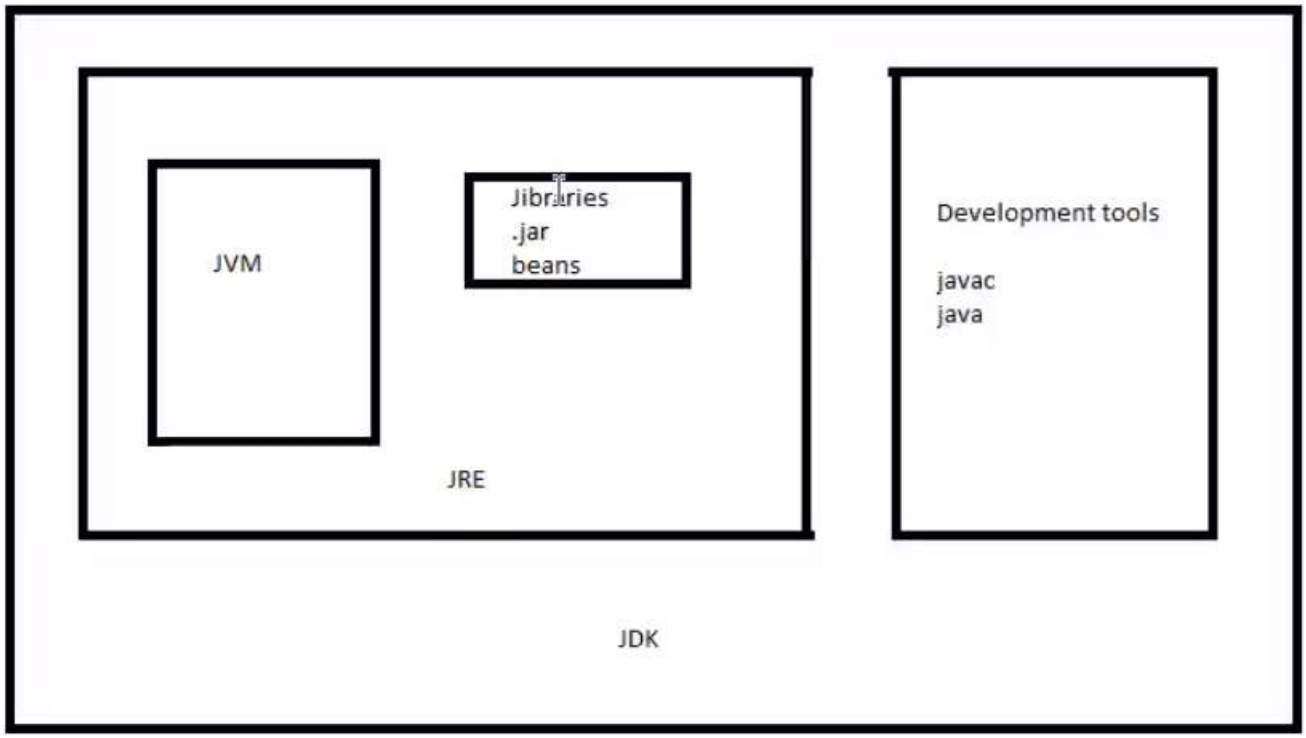
Interface, Wrapper, Exception and Collection

Map of JVM



Interface in Java

An interface in Java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction.

There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also represents the IS-A relationship.

Syntax:

Interface<interface_name>

{

}

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output: Hello

Note: cannot create object of an interface.

- *A class can extend another class and can also implement another interface.*
- *All methods inside an interface are automatically public and abstract. The methods are public because it should be accessible to all implemented classes. Methods are abstract because there is no body for methods.*
- *we can also have variable declaration along with methods in an interface.*
- *An empty interface is permitted. It is called as "Marker Interface or Dagged Interface".*

```
Interface Calculator
{
Void mul();
Void div();
}
Class R2;
{
Public static void main(string args[])
{
Calculator calc = new calculator();
}
}
```

- **Even though the object of the interface cannot be created, reference can be created.**

```
Interface Calculator
{
Void mul();
Void div();
```

```
}
Class R2;
{
Public static void main(string args[])
{
Calculator calc;
}
}
```

- **It is not mandatory for an implementing class to give body for all methods in interface. In such cases we can declare that class as abstract.**

```
Interface Calculator
{
Void mul();
Void div();
}
Abstract class Mycalc implements calculator
{
Public void mul()
{
Int  a=10;
Int  b=20;
Int c=a*b;
System.out.print(c);
Class R3
{
Public static void main(string args[])
}
```

- **An implement class need not have only implemented methods along with that specialized methods can also be there. However using an interface reference we may not be able to access the specialized methods directly, they should be accessed through Downcasting.**

  Eg:

```
Calculator calc =new MyCalculator();
```

(implicit typecasting or upcasting)

```
((MyCalculator)(calc)).add();
```

(explicit typecasting or downcasting)

- **A class implement multiple interfaces because in this case there won't be any diamond shape problem.**
- **An interface cannot implement another interface.**

```
Interface Calculator1
{
Void mul();
Void div();
}
Interface Calculator2 implements Calculator1
{
Class R2;
{
Public static void main(string args[])
}
// Error (Expected body)
```

- **One interface can Extend another interface**

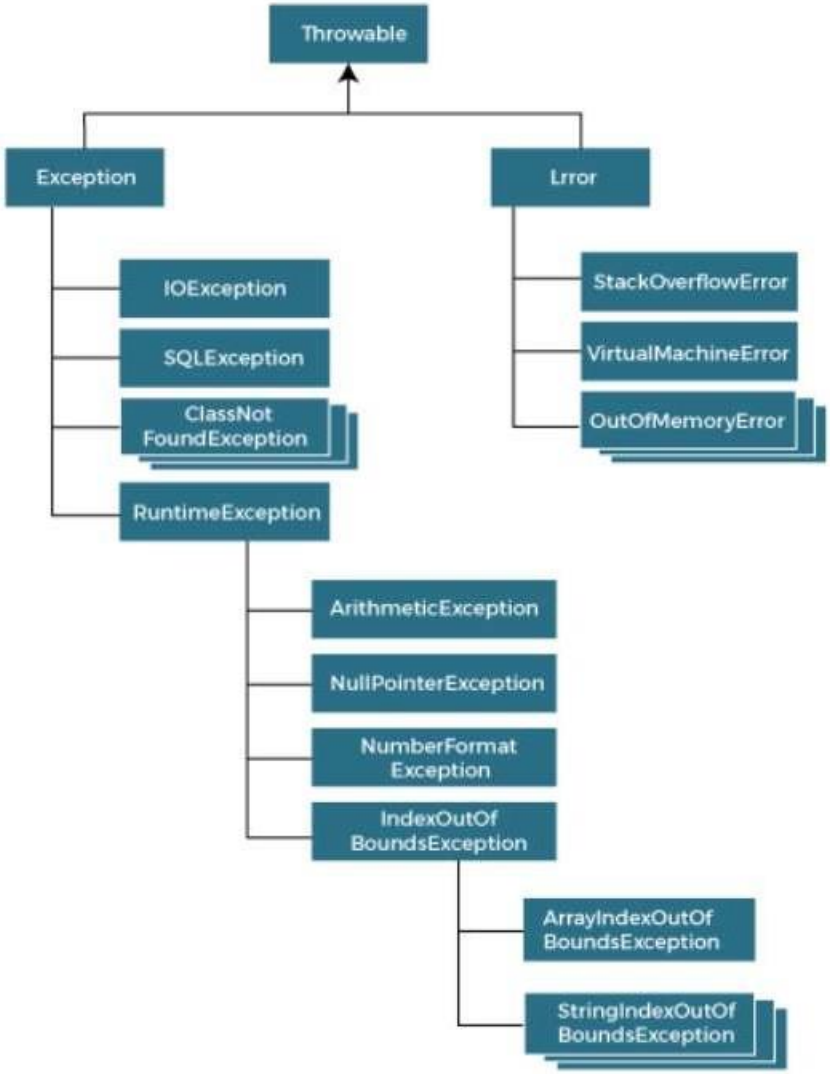
```
Interface Calculator1
{
Void mul();
Void div();
}
Interface Calculator2 extends Calculator1
{
Void mul();
Void div();
}
Class R2;
{
Public static void main(string args[])
{
System.out.println("Hello")
}
}
```

```java
package batch89;

public class Demo extends C implements DemoInterface, NewInterface {
    int abc=10;
    @Override
    public void func(){

    }
    @Override
    public void func2(){

    }
    @Override
    public void fun(){

    }
    public void run(){

    }


    public static void main(String[] args) {

    }
}
```

ERROR vs Exception

Error can't be handle by human, it just happens.

Exception:- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.



Check vs Uncheck Exception
Check exception directly come from exception class. Uncheck exception comes from Runtime Exception.

try-catch
*In below program **try** and **catch** will catch the exception and print a custom exception for us. I basically say try if this expression gives this exception then*

```
try{
    System.out.println(5/0);
}catch(ArithmeticException ex){
    System.out.println("There is an Exception");
}

System.out.println("I am still running");
```

```
There is an Exceptionjava.lang.ArithmeticException: / by zero
I am still running
```

Some common Scenarios:

1. **int a=50/0;//ArithmeticException**

2. **String s=null;**

   System.out.println(s.length());//NullPointerException

3. **String s="abc";**

```
int i=Integer.parseInt(s);//NumberFormatException
```

1. **int a[]=new int[5];**

a[10]=50; //ArrayIndexOutOfBoundsException

Syntax of Java try-catch

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

Syntax of try-finally block

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

Some example of try catch

```
public class TryCatchExample2 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
       //handling the exception
    catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

Output:

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

Example: throwing unchecked Exception

```
public class TestThrow1 {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

Example: throwing checked Exception

```
import java.io.*;
public class TestThrow2 {
   //function to check if person is eligible to vote or not
   public static void method() throws FileNotFoundException {

      FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
      BufferedReader fileInput = new BufferedReader(file);

      throw new FileNotFoundException();
```

```
    }
    //main method
    public static void main(String args[]){
      try
      {
        method();
      }
      catch (FileNotFoundException e)
      {
        e.printStackTrace();
      }
      System.out.println("rest of the code...");
  }
}
```

Output:

----------WRAPPER DATA TYPES----------

Different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.
- **Primitive vs Wrapper**

boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing:
The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
In below example we are doing autoboxing of int.
int a=20;
Integer j=a; //autoboxing, now compiler will write Integer.valueOf(a) internally

Unboxing: vica versa of autoboxing.
In below example we did unboxing converted wrapper integer into primitive type int.

```
package batch89;

public class Demo {

    public static void main(String[] args) {
        Integer i0 = new Integer(25);
        Integer i1 = new Integer(25);
        Integer i2 = 25;
        int e0 = i0;
        int e1 = Integer.valueOf(i1);

        System.out.println(i0);
        System.out.println(i1);
        System.out.println(i2);
        System.out.println(e0);
        System.out.println(e1);
    }
}
```

Example:

```java
public class Wrapperautoboxinunboxing{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;


Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;


System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);


byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;
```

```
java -cp /tmp/oZttzGBpeM WrapperExample3
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

----------COLLECTIONS----------

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only. There are only three methods in the Iterator interface. They are:

public boolean hasNext() --> returns true if the iterator has more elements otherwise it returns false. **public Object next()** --> returns the element and moves the cursor pointer to the next element. **public void remove()** --> removes the last elements returned by the iterator.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```java
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){

ArrayList<String> list=new ArrayList
    <String>();//Creating arraylist
list.add("Amit");//Adding object in
    arraylist
list.add("Sumit");
list.add("Nitin");
list.add("Rohit");
list.add("Nayan");
list.add("Abhijeet");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
java -cp /tmp/oZttzGBpeM TestJavaCollection1
Amit
Sumit
Nitin
Rohit
Nayan
Abhijeet
```

List Interface
It inhibits a list type data structure in which we can store the ordered collection of objects.
Can have duplicate values.
Implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){

List<String> list=new ArrayList<String
    >();//Creating arraylist
list.add("Amit");//Adding object in
    arraylist
list.add("Sumit");
list.add("Nitin");
list.add("Rohit");
list.add("Nayan");
list.add("Abhijeet");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
java -cp /tmp/oZttzGBpeM TestJavaCollection1
Amit
Sumit
Nitin
Rohit
Nayan
Abhijeet
```