Abstract classes in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

- o An abstract class must be declared with an abstract keyword.

- o Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)

- o But, if a class has at least one abstract method, then the class **must** be declared abstract.

- o If a class is declared abstract, it cannot be instantiated.

- o To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.

- o If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

- o It can have constructors and static methods also.

- o It can have final methods which will force the subclass not to change the body of the method.

Abstraction lets you focus on what the object does instead of how it does it.

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

EXAMPLE:

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}

class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();
s.draw();
}
}
```

Java - Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.

- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.

- The byte code of an interface appears in a **.class** file.

- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.

- An interface does not contain any constructors.

- All of the methods in an interface are abstract.

- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is implemented by a class.

- An interface can extend multiple interfaces.

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
   // Any number of final, static fields
   // Any number of abstract method declarations\
}
```

```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void travel();
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.

- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.

- Methods in an interface are implicitly public.

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }

  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

When overriding methods defined in interfaces, there are several rules to be followed −

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.

- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.

- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.

- A class can extend only one class, but implement many interfaces.

- An interface can extend another interface, in a similar way as a class can extend another class.

- An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

- The following Sports interface is extended by Hockey and Football interfaces.

```java
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

```java
public interface Hockey extends Sports, Event
```
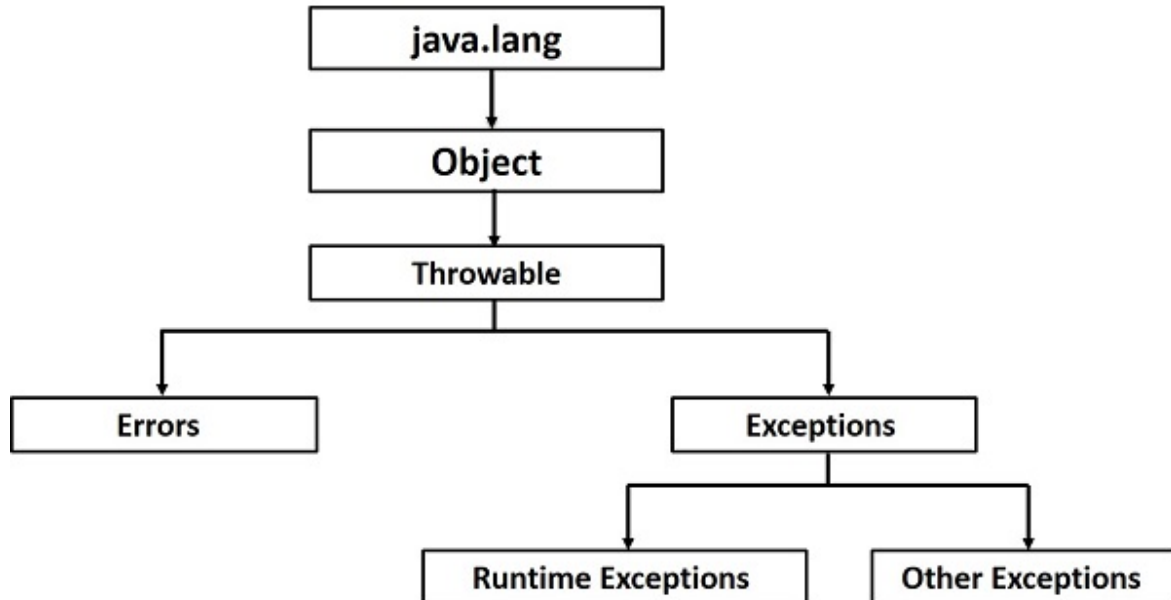
EXCEPTION HANDLING

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. The core advantage of exception handling is **to maintain the normal flow of the application**. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
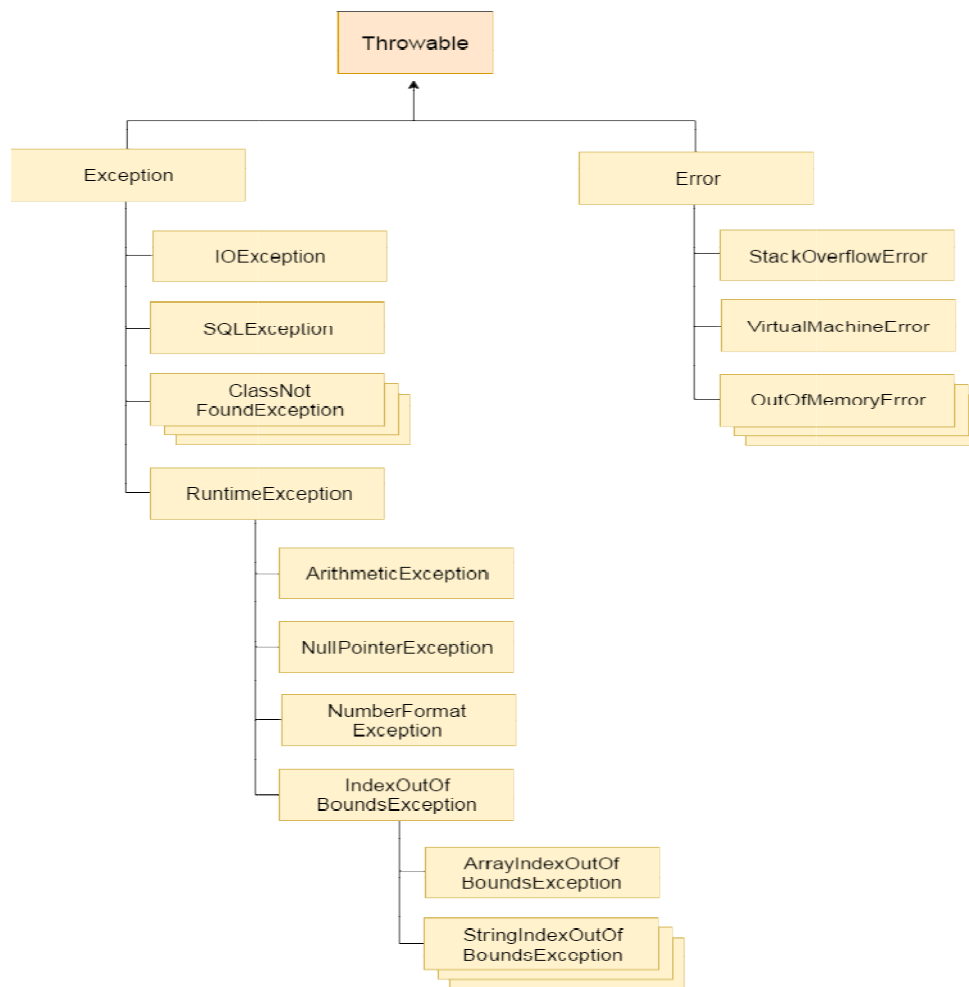
An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.

- A file that needs to be opened cannot be found.

- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

```
                    ┌─────────────────┐
                    │   java.lang     │
                    └────────┬────────┘
                             ↓
                    ┌─────────────────┐
                    │     Object      │
                    └────────┬────────┘
                             ↓
                    ┌─────────────────┐
                    │   Throwable     │
                    └────────┬────────┘
              ┌──────────────┴──────────────┐
              ↓                              ↓
       ┌─────────────┐              ┌─────────────────┐
       │   Errors    │              │   Exceptions    │
       └─────────────┘              └────────┬────────┘
                              ┌──────────────┴──────────────┐
                              ↓                              ↓
                    ┌────────────────────┐      ┌────────────────────┐
                    │ Runtime Exceptions │      │  Other Exceptions  │
                    └────────────────────┘      └────────────────────┘
```

Based on these, we have three categories of Exceptions.

Checked exceptions − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions. The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation. The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Errors − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Java Exception Keywords

try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

```
try {

  // Block of code to try

}

catch(Exception e) {

  // Block of code to handle errors

}
```

Example

```java
public class MyClass {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    }
catch (ArrayIndexOutofBounds e)
 { System.out.println("Something w ent wrong."); }


catch (Exception e) {
    System.out.println("Something went wrong.");
  }

 }
}
```

Example

```java
public class MyClass {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

Example

```
public class MyClass {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    } else {
      System.out.println("Access granted - You are old enough!");
    }
  }

 public static void main(String[] args) {
  checkAge(15);
 }
}
```

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException**
   It is 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. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified

8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
9. **NullPointerException**
   This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**
    This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**
    This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException**
    It is thrown by String class methods to indicate that an index is either negative than the size of the string

/* Java program to demonstrate user defined exception .This program throws an exception whenever balance amount is below Rs 1000 */

```
class MyException extends Exception
{
        //store account information
        private static int accno[] = {1001, 1002, 1003, 1004};

        private static String name[] = {"Nish", "Shubh", "Sush", "Abhi", "Akash"};

        private static double bal[] = {10000.00, 12000.00, 5600.0, 999.00, 1100.55};

        MyException()
        {}

        MyException(String str)
        { super(str); }


public static void main(String[] args)
        {
        try {
                for (int i = 0; i < 5 ; i++)
                {
                System.out.println( accno[i] + "\t" + name[i] + "\t" + bal[i]);
```

```java
        // display own exception if balance < 1000
            if (bal[i] < 1000)
            {   MyException me = new MyException("Balance is less than 1000");
                throw me;
            }
        }
        } //end of try

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