Exception

What is an exception?

- An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

- There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Advantage of exception handling

Exception handling ensures that the flow of the program doesn’t break when an exception occurs.

For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.

By handling we make sure that all the statements execute and the flow of program doesn’t break.

Difference between error and exception

- **Errors** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.
- **Exceptions** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.

Few examples:
NullPointerException – When you try to use a reference that points to null.

ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

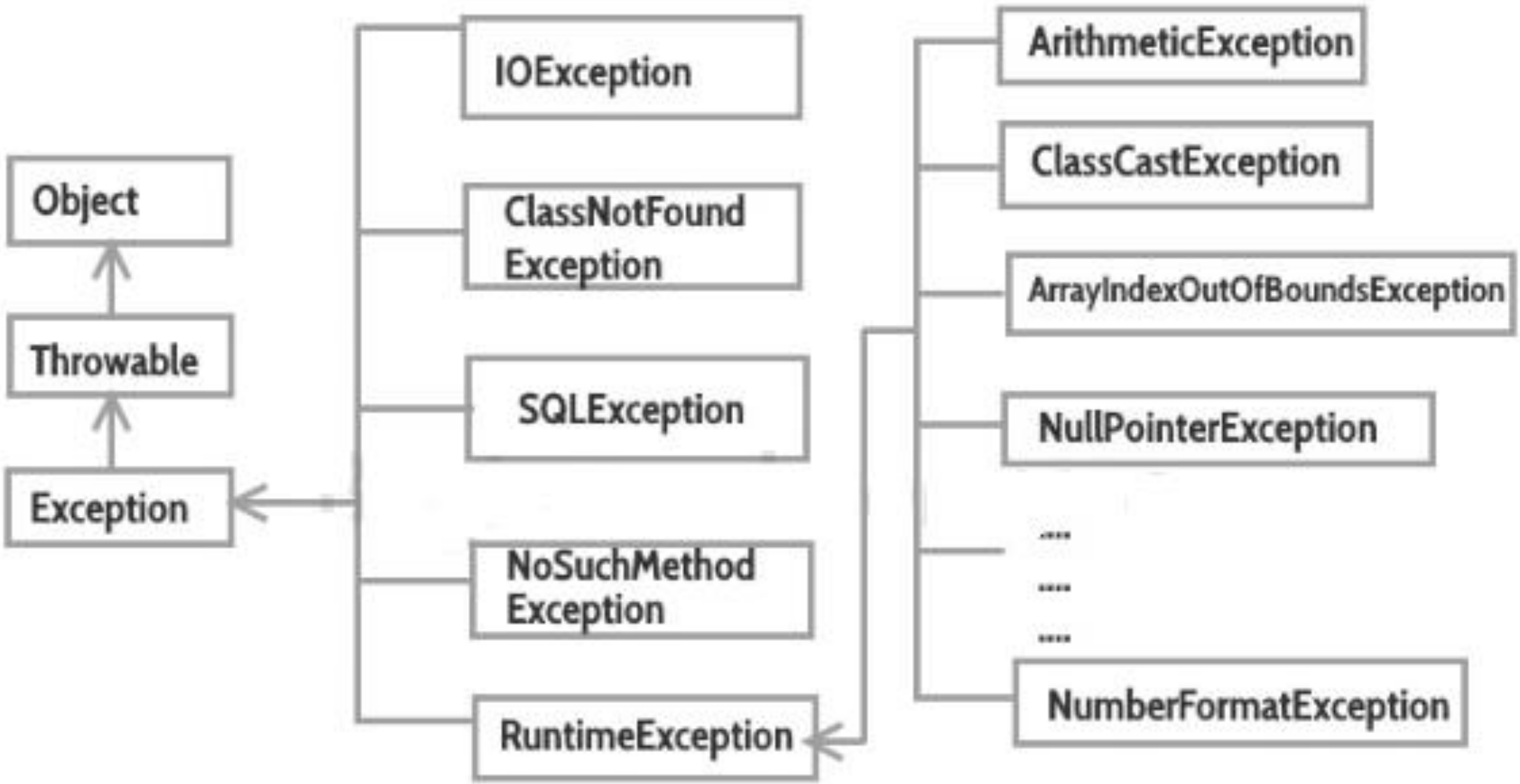
Checked and Unchecked Exceptions

Checked exceptions

- All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, IOException, ClassNotFoundException etc.

Unchecked Exceptions

- Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it’s the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.


Object
Throwable
Exception
IOException
ClassNotFound Exception
SQLException
NoSuchMethod Exception
RuntimeException
ArithmeticException
ClassCastException
ArrayIndexOutOfBoundsException
NullPointerException
....
....
....
NumberFormatException

Try-Catch block

Try block

- The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A **try** block must be followed by **catch** blocks or **finally block or both**.

Catch block

- A catch block is where the exceptions is handled, this block must follow the try block. A single try block can have several catch blocks associated with it. Different exceptions can be handled in different catch blocks.
- When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Example: try catch block

```
class Example1 {
  public static void main(String args[]) {
    int num1, num2;
    try {
 /* We suspect that this block of statement can throw exception so we handled it by placing these statements inside try and
handled the exception in catch block */
      num1 = 0;
      num2 = 62 / num1;
      System.out.println(num2);
      System.out.println("Hey I'm at the end of try block");
    }
    catch (ArithmeticException e) {
      /* This block will only execute if any Arithmetic exception occurs in try block */
      System.out.println("You should not divide a number by zero");
    }
    catch (Exception e) {
      /* This is a generic Exception handler which means it can handle all the exceptions. This will execute if the exception is
not handled by previous catch blocks.*/
      System.out.println("Exception occurred");
    }
    System.out.println("I'm out of try-catch block in Java.");
  }
}
```

Output:
You should not divide a number by zero
I'm out of try-catch block in Java.

Example: Multiple catch blocks

```
class Example2{
  public static void main(String args[]){
    try{
       int a[]=new int[7];
       a[4]=30/0;
       System.out.println("First print statement in try block");
    }
    catch(ArithmeticException e){
      System.out.println("Warning: ArithmeticException");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Warning: ArrayIndexOutOfBoundsException");
    }
    catch(Exception e){
      System.out.println("Warning: Some Other exception");
    }
   System.out.println("Out of try-catch block...");
  }
}
```

Output:
Warning: ArithmeticException
Out of try-catch block....

Example: Array out of bound

```
class Example{
   public static void main(String args[]){
      try{
         int arr[]=new int[7];
         arr[10]=10/5;
         System.out.println("Last Statement of try block");
      }
      catch(ArithmeticException e){
         System.out.println("You should not divide a number by zero");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Accessing array elements outside of the limit");
      }
      catch(Exception e){
         System.out.println("Some Other Exception");
      }
      System.out.println("Out of the try-catch block");
   }
}
```

Output:
Accessing array elements outside of the limit
Out of the try-catch block

Generic exception catch block should be placed at the end of all other specific exception catch block

The **finally** block

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

- **Syntax of Finally block**

```
try {
    //Statements that may cause an exception
}
catch {
   //Handling exception
}
finally {
   //Statements to be executed
}
```

```java
class Example
{
   public static void main(String args[]) {
      try{
         int num=121/0;
         System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("Number should not be divided by zero");
      }
      /* Finally block will always execute
       * even if there is no exception in try block
       */
      finally{
         System.out.println("This is finally block");
      }
      System.out.println("Out of try-catch-finally");
   }
}
```

Example: Try catch finally blocks

Output:

```
Number should not be divided by zero
This is finally block
Out of try-catch-finally
```

Important points of finally block

- A finally block must be associated with a try block. Those statements that must be always executed should be placed in this block.
- Finally block is optional, a try-catch block is sufficient for exception handling, however if a finally block is placed then it will always run after the execution of try block.
- In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.
- An exception in the finally block, behaves exactly like any other exception.
- The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.

```java
class Example1{
  public static void main(String args[]){
    try{
       System.out.println("First statement of try block");
       int num=45/3;
       System.out.println(num);
    }
    catch(ArrayIndexOutOfBoundsException e){
       System.out.println("ArrayIndexOutOfBoundsException");
    }
    finally{
       System.out.println("finally block");
    }
    System.out.println("Out of try-catch-finally block");
  }
}
```

Example: Try catch finally blocks

Output:

```
First statement of try block
15
finally block
Out of try-catch-finally block
```

```java
class Example3{
   public static void main(String args[]){
      try{
         System.out.println("First statement of try block");
         int num=45/0;
         System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("ArithmeticException");
      }
      finally{
         System.out.println("finally block");
      }
      System.out.println("Out of try-catch-finally block");
   }
}
```

Example2: Try catch finally blocks

Output:

```
First statement of try block
ArithmeticException
finally block
Out of try-catch-finally block
```

throw exception in java

- We can define our own set of conditions or rules and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword.

Example: throw exception

```java
/* In this program we are checking the Student age
 * if the student age<12 and weight <40 then our program
 * should return that the student is not eligible for registration.
 */
public class ThrowExample {
   static void checkEligibilty(int stuage, int stuweight){
      if(stuage<12 && stuweight<40) {
         throw new ArithmeticException("Student is not eligible for registration");
      }
      else {
         System.out.println("Student Entry is Valid!!");
      }
   }

   public static void main(String args[]){
     System.out.println("Welcome to the Registration process!!");
     checkEligibilty(10, 39);
     System.out.println("Have a nice day..");
 }
}
```

Output:
Welcome to the Registration process!!Exception in thread "main" java.lang.ArithmeticException: Student is not eligible for registration at packagename.ThrowExample.checkEligibilty(ThrowExample.java:9) at packagename.ThrowExample.main(ThrowExample.java:18)

Throws clause in java

- By using throws we can declare multiple exceptions in one go.
- Suppose a method myMethod() has statements that can throw either ArithmeticException or NullPointerException, the try-catch statements shown below can be used:

```
public void myMethod()
{
  try {
    // Statements that might throw an exception
  }
  catch (ArithmeticException e) {
    // Exception handling statements
  }
  catch (NullPointerException e) {
    // Exception handling statements
  }
}
```

Throws clause in java (cont)

- But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable.
- Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the checked exceptions that are declared using throws, must be handled when you are calling this method else you will get compilation error.
- One more usage of throws keyword is to allow checked exception to propagate in the calling chain.
- However, for unchecked Exceptions, they are by default forwarded in calling chain (propagated).

Reference: www.geeksforgeeks.org

```java
import java.io.*;
class ThrowExample {
  void myMethod(int num)throws IOException, ClassNotFoundException{
     if(num==1)
        throw new IOException("IOException Occurred");
     else
        throw new ClassNotFoundException("ClassNotFoundException");
  }
}

public class Example1{
  public static void main(String args[]){
   try{
     ThrowExample obj=new ThrowExample();
     obj.myMethod(1);
   }catch(Exception ex){
     System.out.println(ex);
    }
  }
}
```

Example: throws clause in Java

Output:

java.io.IOException: IOException Occurred

Propagate with Unchecked Exceptions

```
public class PropagateUncheck {
void m()
   {
       int data = 50 / 0; // unchecked exception occurred
   }
void n()
   {
      m();
   }
void p()
   {
      try {
         n();
      }
      catch (Exception e) {
         System.out.println("Exception handled");
      }
   }
```

```
 public static void main(String args[])
   {
      PropagateUncheck obj = new PropagateUncheck();
      obj.p();
      System.out.println("Normal flow...");
   }
}
```

Output:

Exception handled
Normal flow...

Propagate with Checked Exception

```
import java.io.IOException;
public class PropagateCheck {

void m() throws IOException
   {
    throw new IOException("device error");
   }
 void n() throws IOException
   {
      m();
   }
   void p()
   {
      try {
       n();
      }
      catch (Exception e) {
         System.out.println("exception handled");
      }
   }
```

```
   public static void main(String args[])
   {
      PropagateCheck obj = new PropagateCheck();
      obj.p();
      System.out.println("normal flow...");
   }
}
```

Output:

Exception handled
Normal flow...

Difference between throw and throws

Reference:www.spiroprojects.com

throw	throws
1. Java throw keyword is used to explicitly throw an exception	1. Java throws keyword is used to declare an exception.

2. void m(){
throw new
ArithmeticException("sorry");
}
	2. void m()throws ArithmeticException{
	//method code
	}

3. Checked exception cannot be propagated using throw only.	3. Checked exception can be propagated with throws.
4. Throw is followed by an instance.	4. Throw is followed by a class.
5. Throw is used within the method.	5. Throws is used with the method signature.
6. You cannot throw multiple exceptions.	6. You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.