Chapter 14

Exceptions

ENG : Qusay khudair

Exceptions in Java

Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved.

Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

What are Java Exceptions?

- In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.
- When an exception occurs within a method, it creates an object, This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors

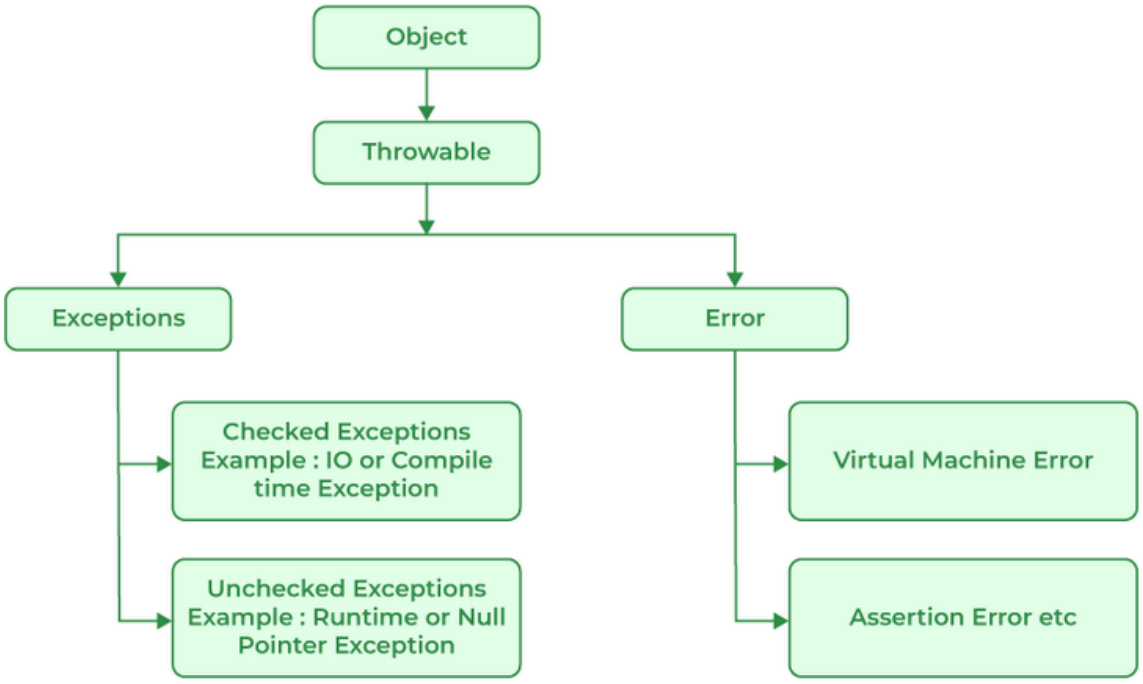
- Out of bound
- Null reference
- Type mismatch
- Opening an unavailable file
- Database errors
- Arithmetic errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Difference between Error and Exception

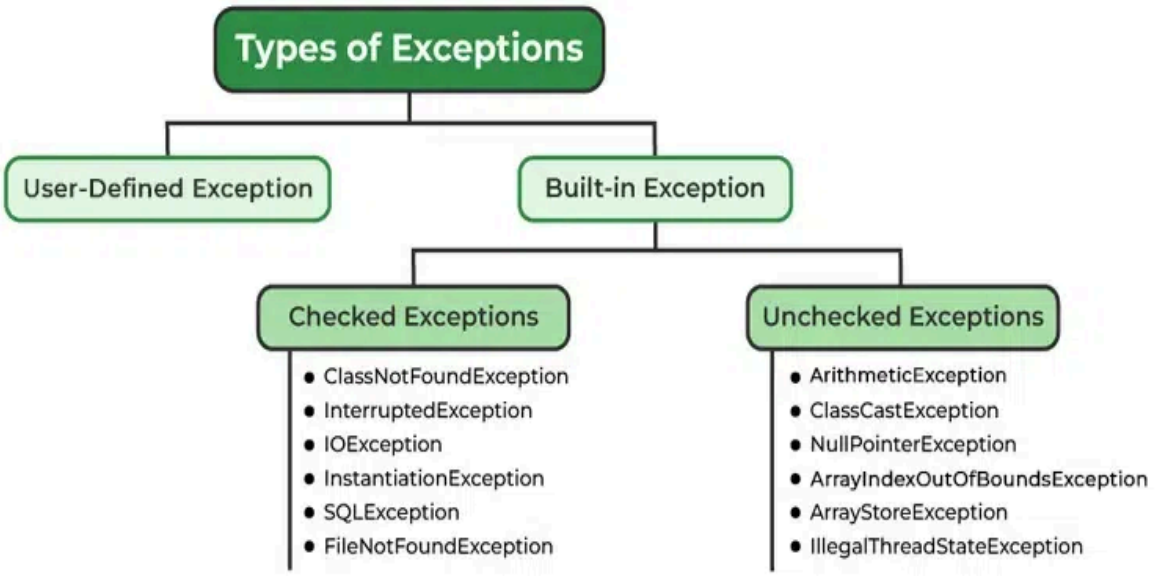
- **Error: An Error indicates a serious problem that a reasonable application should not try to catch.**
- **Exception: Exception indicates conditions that a reasonable application might try to catch.**

Exception Hierarchy



- All exception and error types are subclasses of the class Throwable, which is the base class of the hierarchy. One branch is headed by Exception.
- This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.
- Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.

Types of Exceptions



Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

Exceptions can be categorized in two ways

1. **Built-in Exceptions**
   - **Checked Exception (compile-time) : Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.**
   - **Unchecked Exception : The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.**

- **Here are some examples of unchecked exceptions in Java:**
- ***ArrayIndexOutOfBoundsException: This exception is thrown when you attempt to access an array index that is out of bounds***

- ***NullPointerException: This exception is thrown when you attempt to access a null object reference.***

- ***ArithmeticException: This exception is thrown when you attempt to divide by zero or perform an invalid arithmetic operation.***

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

Methods to print the Exception information

1. **printStackTrace() and toString() methods**

This method prints exception information in the format of the Name of the exception: description of the exception, stack trace.

Example:

```
import java.io.*;

class Exp{

    public static void main (String[] args) {
      int a=5;
      int b=0;
        try{
          System.out.println(a/b);
        }
      catch(ArithmeticException e){
        e.printStackTrace();}}}
```

Output

```
java.lang.ArithmeticException: / by zero
at GFG.main(File.java:10)
```

2. **getMessage()**

The getMessage() method prints only the description of the exception.

Example:

```
import java.io.*;

class Exp{
    public static void main (String[] args) {
      int a=5;
      int b=0;
        try{
          System.out.println(a/b);
        }
      catch(ArithmeticException e){
        System.out.println(e.getMessage()); }}}
```

Output

```
/ by zero
```

How Programmer Handle an Exception?

Customized Exception Handling: Java exception handling is managed via keywords: try, catch, throw, throws, and finally.

Throw and Throws Keyword

Benchmark	Throw Keyword	Throws Keyword
Usage	You use the Throw keyword to throw an exception explicitly.	You can use the Throws keyword to declare that a method might throw an exception.
Declaration	The Throw keyword is used inside a method.	The Throws keyword is used in the method signature.
The syntax is followed by	An instance of exception to be thrown.	Class names of exceptions to be thrown.
Can this keyword propagate the checked exceptions?	No.	Yes.
Number of exceptions that can be thrown using this keyword	Only one exception	Multiple exceptions

The throw keyword : in Java is used to explicitly throw an exception (checked or unchecked) from a method or any block of code. The throw keyword is mainly used to throw custom exceptions.

- **The flow of execution of the program :**

  1. Stops immediately after the throw statement is executed .

  2. The nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception.

  3. If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked, and so on.

  4. If no matching catch is found then the default exception handler will halt the program.

Ex 1 :

```
class ThrowExcep {
    static void fun()
    {
        try {
            throw new NullPointerException("demo");
        }
        catch (NullPointerException e) {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try {
            fun();
        }
        catch (NullPointerException e) {
            System.out.println("Caught in main.");
        }
    }
}
```

Output

```
Caught inside fun().
Caught in main.
```

Ex 2:

```
public class TestThrow {
  public static void checkNum(int num) {
    if (num < 1) {
            throw new ArithmeticException("\nNumber is negative, cannot
    calculate square");
        }
         else {
            System.out.println("Square of " + num + " is " + (num*num));    } }
```

```
        public static void main(String[] args) {
            TestThrow obj = new TestThrow();
            obj.checkNum(-3);
            System.out.println("Rest of the code..");
        }
    }
```

Output:

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

C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

The throws keyword : in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax of Java throws

```
- Data type Method_name(parameters) throws exception_list
```

`exception_list is a comma separated list of all the exceptions which a method might throw.`

In a program, if there is a chance of raising an exception then the compiler always warns us about it and compulsorily we should handle that checked exception, Otherwise, we will get compile time error saying unreported exception ….. must be caught or declared to be thrown. To prevent this compile time error we can handle the exception in two ways:

1.By using try - catch

2.By using the throws keyword

```
public class TestThrows {
        public static int divideNum(int m, int n) throws ArithmeticException {
           int div = m / n;
           return div;  }
        public static void main(String[] args) {
           TestThrows obj = new TestThrows();
           try {
              System.out.println(obj.divideNum(45, 0));   }
           catch (ArithmeticException e){
              System.out.println("\nNumber cannot be divided by 0");  }
           System.out.println("Rest of the code..");    } }
```

Output:

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

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

Flow control in try - catch - finally in Java

- In Java, try-catch-finally blocks manage the flow of exceptions and ensure that certain code executes regardless of what happens:

1. try Block: Contains code that may throw an exception. If an exception occurs, execution jumps to the catch block.
2. catch Block: Catches and handles exceptions. There can be multiple catch blocks for different exception types.
3. finally Block: Always executes after try and catch, even if an exception occurs or a return statement is used. It's typically used for resource cleanup.

- **Key Points:**

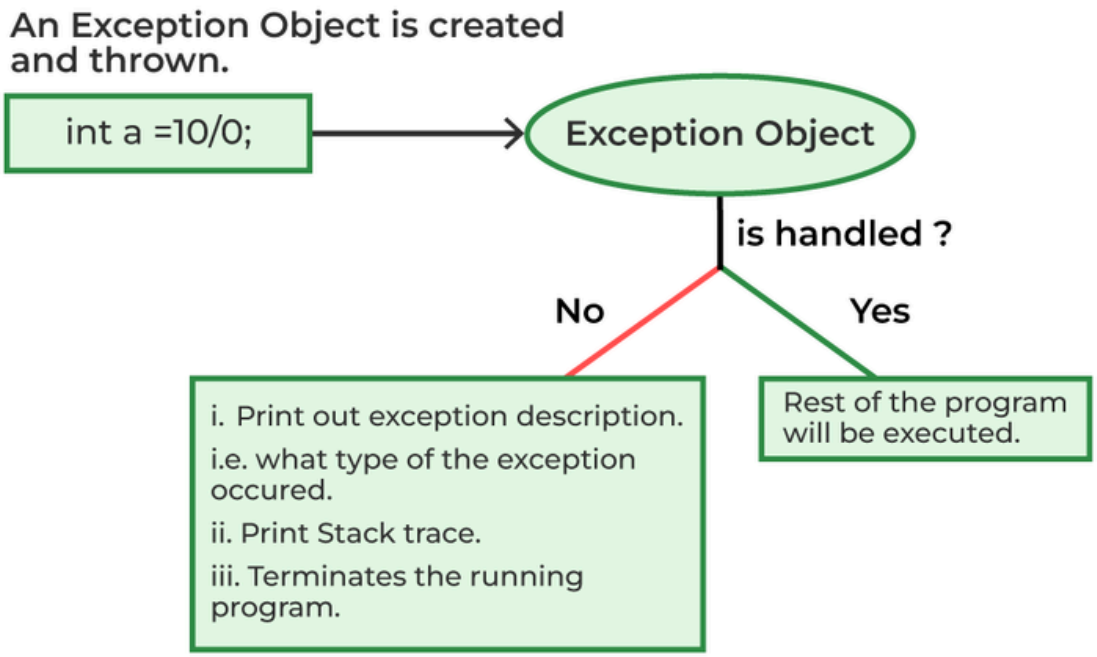
- **Normal Flow:** Executes **try**, then **finally** if no exception.
- **Exception Flow:** Jumps from **try** to **catch**, then executes **finally**.
- **Resource Management:** **finally** ensures resources are closed or cleaned up.

```
try {
   // Code that might throw an exception
} catch (ExceptionType e) {
   // Handling the exception}
finally {
   // Code that always executes
}
```

- **Advanced Flow Control:**

- **Return Statements:** Even if **try** or **catch** contains a return statement, **finally** executes before the method returns.
- **Exceptions in finally:** If **finally** throws an exception, it can override exceptions thrown in **try** or **catch**.

The summary is depicted via visual aid below as follows:



Creating Custom Exception Classes in Java

In Java, you can create custom exceptions to handle specific error conditions tailored to your application's needs.

Custom exceptions provide more precise control over error handling and can help make your code more readable and maintainable.

Creating a Custom Exception Class

To create a custom exception class, follow these steps:

Inherit from `Exception` or `RuntimeException`:

- If you want your custom exception to be a checked exception (one that must be caught or declared in the method signature), extend `Exception`.
- If you prefer your custom exception to be an unchecked exception (one that does not need to be explicitly handled), extend `RuntimeException`.

Define the Custom Exception Class:

```
// For a checked exception
public class MyCheckedException extends Exception {
    // Default constructor
    public MyCheckedException() {
        super();
    }
```

```
    // Constructor that accepts a message
    public MyCheckedException(String message) {
        super(message);
    }
    // Constructor that accepts a message and a cause
    public MyCheckedException(String message, Throwable cause) {
        super(message, cause);
    }
    // Constructor that accepts a cause
    public MyCheckedException(Throwable cause) {
        super(cause);
    }
}
// For an unchecked exception
public class MyUncheckedException extends RuntimeException {
    // Default constructor
    public MyUncheckedException() {
        super();
    }
    // Constructor that accepts a message
    public MyUncheckedException(String message) {
        super(message);
    }
    // Constructor that accepts a message and a cause
    public MyUncheckedException(String message, Throwable cause) {
        super(message, cause);}
    // Constructor that accepts a cause
    public MyUncheckedException(Throwable cause) {
        super(cause); }}
```

Throwing the Custom Exception: You can throw your custom exception using the `throw` keyword:

```
public void someMethod() throws MyCheckedException {
    // Some condition that triggers the exception
    if (someCondition) {
        throw new MyCheckedException("An error occurred");
    }
}
```

Catching the Custom Exception: Handle the custom exception using a `try-catch` block:

```
try {
    someMethod();
} catch (MyCheckedException e) {
    System.out.println("Caught exception: " + e.getMessage());
}
```

Declaring New Exception Types

- **Extend existing exception class :**

```
public class TooSmallException extends ArithmeticException {

      public TooSmallException (  ){
             super ("Value is less than 100");
}

      public TooSmallException (String message){
             super (message);
       }
}
```

```
public class TestTooSmallException{
public static int  compute(int x, int y) throws TooSmallException{
if(x<100 || y<100)
throw new TooSmallException();
else
return x+y;}


public static void main(String args[]) {
try {
System.out.println(compute(20,150));

}catch(TooSmallException e) {
System.out.println(e);
   }
        }
}
```

System Errors in Java

- **System errors in Java are serious issues that occur at the JVM level, usually due to resource exhaustion or internal system failures.**

- **These errors are represented by the Error class and its subclasses, and they indicate conditions that a typical application cannot recover from.**

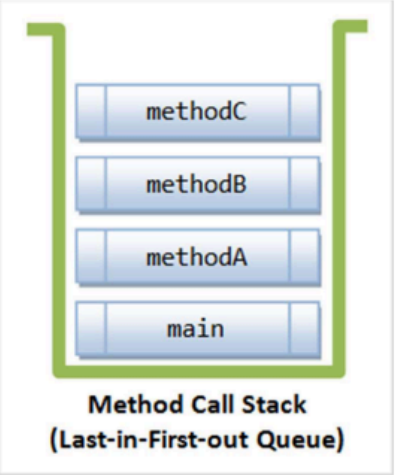
- **OutOfMemoryError: Occurs when the JVM runs out of memory.**
- **StackOverflowError: Happens when the stack overflows, typically due to deep or infinite recursion.**
- **InternalError: Indicates an internal problem with the JVM.**
- **VirtualMachineError: Represents an error with the Java Virtual Machine.**

Ex :

```
public class Example {
    public static void main(String[] args) {
        try {
            infiniteRecursion(); // This will cause StackOverflowError
        } catch (StackOverflowError e) {
            System.out.println("Caught: " + e);
        }
    }

    public static void infiniteRecursion() {
        infiniteRecursion(); // This causes the stack to overflow }}
```

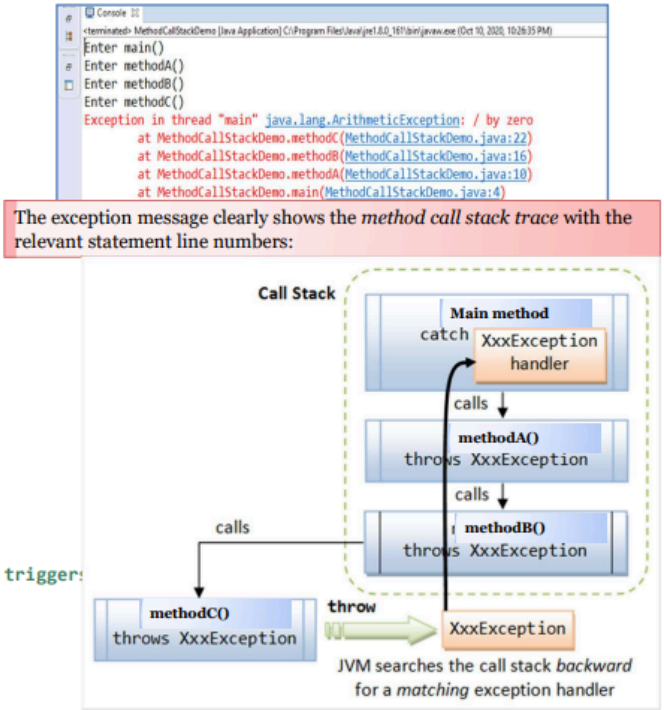
Exception and Method Call Stack

```
public class MethodCallStackDemo {
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit main()");
}

   public static void methodA() {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit methodA()");}

   public static void methodB() {
      System.out.println("Enter methodB()");
      methodC();
      System.out.println("Exit methodB()");}

   public static void methodC() {
   System.out.println("Enter methodC()");
   System.out.println(1 / 0);  // divide-by-0 triggers an
ArithmeticException
   System.out.println("Exit methodC()");}


}
```



Exception and Call Stack

```
1.  public class MethodCallStackDemo {
2.     public static void main(String[] args) {
3.        System.out.println("Enter main()");
4.        methodA();
5.        System.out.println("Exit main()");
6.  }
7.
8.     public static void methodA() {
9.        System.out.println("Enter methodA()");
10.       methodB();
11.       System.out.println("Exit methodA()");
12. }
13.
14.    public static void methodB() {
15.       System.out.println("Enter methodB()");
16.       methodC();
17.       System.out.println("Exit methodB()");
18. }
19.
20.    public static void methodC() {
21.    System.out.println("Enter methodC()");
22.    System.out.println(1 / 0);  // divide-by-0 trigger
23.    System.out.println("Exit methodC()");}
24.
25.
26. }
```



- **For More … Follow Me :**

  Eng.Qusay Khudair

- **For Business :**

  Linkedin

  Behance

  Github

- **For My Files and Slides :**

  studocu.com

- **Whatsapp : +972567166452**

AUG - PLUS 2024