Java - Exceptions

An exception (or exceptional event) **is a problem that arises during the execution of a program**. 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.

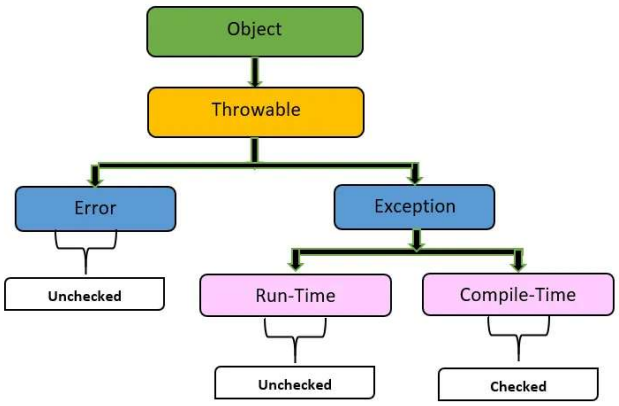
- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

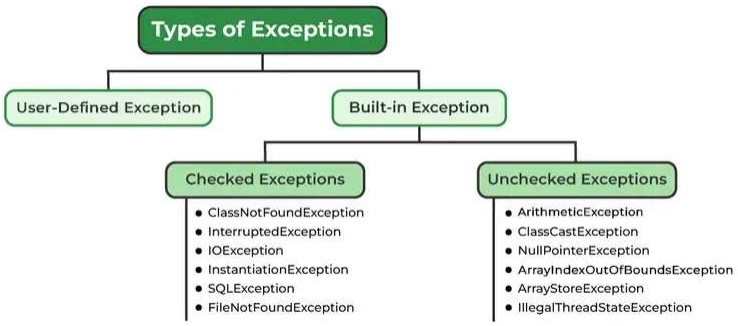
Some of these exceptions are caused by user error, others by **programmer error**, and others by **physical resources that have failed in some manner**.

Java Exception Categories

Based on these, we have the following categories of Exceptions. You need to understand them to know how exception handling works in Java.

- Checked exceptions
- Unchecked exceptions
- Errors





1. Checked Exceptions:

Checked exceptions, also known as **compile-time exceptions**, are exceptions that must be either caught or declared in the method signature using the **throws keyword**. These exceptions are typically **used to handle expected error scenarios that a program can recover from**. They force the developer to acknowledge and handle these exceptional conditions, ensuring that appropriate actions are taken.

Some common examples of **checked** exceptions in Java include:

- IOException: Thrown when an input or output operation fails, such as file I/O errors.
- SQLException: Thrown when there is an issue with database connectivity or queries.
- FileNotFoundException: Thrown when attempting to access a file that does not exist.
- ClassNotFoundException: Thrown when the Java runtime cannot find a specified class.

Examples

1.

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter integer: ");
int number = scanner.nextInt();
```

What would happen if we entered, say, abc123, an input value that is not an int? We would get an error message like this:

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:819)
at java.util.Scanner.next(Scanner.java:1431)
at java.util.Scanner.nextInt(Scanner.java:2040)
at java.util.Scanner.nextInt(Scanner.java:2000)
at Ch8Sample1.main(Ch8Sample1.java:35)
```

This **error message indicates that the system has caught an exception called the InputMismatchException**, an error that occurs when we try to convert a string that cannot be converted to a numerical value.

2.

```
Public class Main {
  Public static void main (String[] args) {
  int a;
  a = "this is incompatible type, 'a' should be String"; } }
```

We would get an error message like this:

```
Exception in thread "main" java.lang.RuntimeException:
Uncompilable source code - incompatible types:
java.lang.String cannot be converted to int
```

Java Unchecked Exceptions

An unchecked exception is an exception that occurs at the **time of execution**. These are also called **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.

Example: Unchecked Exceptions in Java

For example, if you have **declared an array of size 5 in your program** and try to call the 6 element of the array then an **ArrayIndexOutOfBoundsExceptionexception** occurs.

```
public class Unchecked_Demo {
  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]); }}
```

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

Java 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.

Examples of errors:

- **OutOfMemoryError**: Raised when the JVM runs out of memory.
- **StackOverflowError**: Raised when the call stack of a program exceeds its limit.
- **NoClassDefFoundError**: Raised when the JVM cannot find a class definition.

Java Exception Handling

In Java, **exception handling is a mechanism to handle runtime errors**, allowing the **normal flow of a program to continue**. Exceptions are events that occur during program execution that disrupt the normal flow of instructions.

Basic try-catch Example

- The try block contains code that might throw an exception,
- The catch block handles the exception if it occurs.

Syntax

```
try {
  // Block of code to try
}
catch(Exception e) {
  // Block of code to handle errors
}
```

Example

```
class G{
   public static void main(String[] args) {
      int n = 10;
      int m = 0;

      try {
         int ans = n / m;
         System.out.println("Answer: " + ans); }
      catch (ArithmeticException e){
         System.out.println("Error: Division by 0!");  } } }
```

Output

Error: Division by 0!

1. What will be displayed on the console window when the following code is executed, and the user enters abc123 and 14?

```
import java.util.Scanner;
import java.util.InputMismatchException;

Scanner scanner = new Scanner(System.in);
try {
  int num1 = scanner.nextInt();
  System.out.println("Input 1 accepted");
  int num2 = scanner.nextInt();
  System.out.println("Input 2 accepted");}
catch (InputMismatchException e) {
 System.out.println("Invalid Entry");}
```

2. What is wrong with the following code? It attempts to loop until the valid input is entered.

```
Scanner scanner = new Scanner(System.in);
try {
  while (true) {
  System.out.print("Enter input: ");
  int num = scanner.nextInt();}}
```

```
catch (InputMismatchException e) {
scanner.next();
System.out.println("Invalid Entry"); }
```

throw and throws Keywords

1. throw: Used to explicitly **throw a single exception**. We use throw when something goes wrong (or "shouldn't happen") and we **want to stop normal flow** and hand control to exception handling.

Syntax:

```
throw new ExceptionType ("Error message");
```

Usage: It appears **inside the method body**.
Purpose: To manually trigger an exception object and transfer control to the nearest try-catch block or the default exception handler.
Exception Types: Can throw both **checked and unchecked exceptions**.

Example

```
class Demo {
   static void checkAge(int age) {

      if (age < 18) {
         throw new ArithmeticException("Age must be 18 or above");  } }

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

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Age must be 18 or
above
at Demo.checkAge(Demo.java:5)
at Demo.main(Demo.java:11)
```

Example

```
// Throwing an arithmetic exception
class G {
   public static void main(String[] args){
      int numerator = 1;
      int denominator = 0;
```

```
 if (denominator == 0) {
    // Manually throw an ArithmeticException
    throw new ArithmeticException("Cannot divide by zero"); }
 else {
    System.out.println(numerator / denominator);   }}}
```

Output:

Hangup (SIGHUP)
Exception in thread "main" java.lang.ArithmeticException: **Cannot divide by zero**
    at Geeks.main(Geeks.java:9)

2. throws: Declares exceptions that a method might throw, informing the caller to handle them. It is **mainly used with checked exceptions** (explained below). If a method calls another method that throws a checked exception, and it doesn't catch it, it must declare that exception in its throws clause.

Note:

When handling exceptions without the use of a try-and-catch block, this keyword is used. It **does not handle the exceptions that a method can throw at the caller**; it **only specifies them**.

Syntax:

returnType methodName() **throws exception_list** { }

Usage: It appears in the **method declaration/signature**, after the method parameters.
Purpose: To inform the caller about **potential exceptions**, allowing them to provide a **try-catch block or declare throws themselves**.
Exception Types: It is mainly used for **checked** exceptions (like **IOException or ClassNotFoundException**), which must be caught or declared. **Unchecked** exceptions (like RuntimeException subclasses) do not require a throws declaration.

Example

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {
  static void checkAge(int age) throws ArithmeticException {
    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); // Set age to 15 (which is below 18...)
  }
}
```

Example

```
class Demo {
 public static void checkAge (int age) throws Exception{
       if(age > 63)
          throw new Exception("you are too old!");        }

      public static void main(String[] args) {
      try {
          checkAge(70); }

       catch( Exception e ) {
  // getMessage is used to retrieve the specific detail message associated with
a Throwable instance
          System.out.println("you are is "+e.getMessage() );   }  }}
```

Example

```
public static void main(String[] args) throws Exception {
int x = 1;
int y = 2;
System.out.println("A");
System.out.println("B");
if (x < y) throw new Exception("Error");
System.out.println("C");}}
```

Output:

```
A
B
```

Exception in thread "main" java.lang.Exception: Error at ExceptionTest.main(ExceptionTest.java:8)

Notice that “C” was never printed and the program halted

Example

```
public static void checkAge (int age) throws ArithmeticException, Exception{
      if(age <= 0) {
         throw new ArithmeticException("This is ArithmeticException");        }
      else{
         System.out.println( 100/age );  }    }

        public static void main(String[] args) {
              try {
                 checkAge(0);              }
              catch( ArithmeticException e1 ) {
                 System.out.println( e1.getMessage() );     }
              catch( Exception e2 ) {
                 System.out.println( e2.getMessage() ); }}}
```

Output

The result is : This is ArithmeticException

Example

```
import java.io.IOException;

class Test {
static void Nmae (String name) throws IOException{
                if (name == "")
                        throw new IOException("Empty name");    }
public static void main(String[] args) throws IOException {
Nmae (""); }}
```

Output

Exception in thread "main" java.io.IOException: Empty name

Unhandled Exception

```
class G {
   public static void main(String[] args)
// pause the execution of the currently running thread for a specified period
   {  Thread.sleep(10000);
      System.out.println("Hello Ges");  }}
```

Output:

error: unreported exception **InterruptedException**; must be caught or declared to be thrown

Explanation: In the above program, we are getting **compiled time error** because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute the main() method which will cause InterruptedException.

Example: Using throws to Handle Exception

```
class G {
   public static void main(String[] args) throws InterruptedException
   {
      Thread.sleep(10000);
      System.out.println("Hello Ges"); }}
```

Output:

Hello Ges

Explanation: In the above program, by using the **throws** keyword we handled the **InterruptedException** and we will get the output as **Hello Ges.**

Exercise 1: Write a Java program that attempts to divide an integer by zero, which causes an ArithmeticException, and handle this **exception using a try-catch** block.

```
public class ExceptionExercise1 {
   public static void main(String[] args) {
      try {
         int numerator = 10;
         int denominator = 0;
         // This line will throw an ArithmeticException
         int result = numerator / denominator;
         System.out.println("Result: " + result); } // This line will be skipped
       catch (ArithmeticException e) {
         // The code in this block executes if an ArithmeticException occurs
         System.err.println("Error: Division by zero is not allowed.");
         System.err.println("Exception message: " + e.getMessage());}}}
```

Output:

Error: Division by zero is not allowed.
Exception message: / by zero

Exercise 2: Write a Java program that demonstrates **accessing an array** with an invalid index, handling the **ArrayIndexOutOfBoundsException**, and show how a **general Exception** handler can catch a broader range of issues (**Handling Multiple Exceptions)**.

```
public class ExceptionExercise2 {
   public static void main(String[] args) {
      int[] numbers = {1, 2, 3};
      try {
         // Attempting to access an invalid index (index 10)
         System.out.println(numbers[10]); }
       catch (ArrayIndexOutOfBoundsException e) {
         // Handles the specific array index issue
        System.err.println("Exception caught: Array index is out of bounds.");}
       catch (Exception e) {
         // A more general handler for any other unexpected exception
        System.err.println("An unexpected error occurred: " + e.getMessage());}
      System.out.println("Program continues after handling the exception.");  }}
```

Output:

Exception caught: Array index is out of bounds.
Program continues after handling the exception.