**Q.No:1**

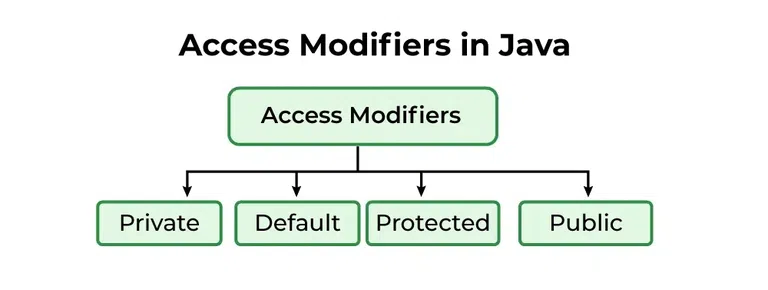
**Access Modifiers in Java**

in Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element. Let us learn about Java Access Modifiers, their types, and the uses of access modifiers in this article.

**Types of Access Modifiers in Java**

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public



**1. Default Access Modifier**

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**.

In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.

**Program 1:**

* Java

|  |
| --- |
| // Java program to illustrate default modifier  **package** p1;    // Class Geek is having Default access modifier  **class** Geek  {  **void** display()      {          System.out.println("Hello World!");      }  } |

**Program 2:**

* Java

|  |
| --- |
| // Java program to illustrate error while  // using class from different package with  // default modifier  **package** p2;  **import** p1.\*;    // This class is having default access modifier  **class** GeekNew  {  **public** **static** **void** main(String args[])      {          // Accessing class Geek from package p1          Geek obj = **new** Geek();            obj.display();      }  } |

**Output:**

Compile time error

**2. Private Access Modifier**

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

* Any other **class of**the **same package will not be able to access** these members.
* Top-level classes or interfaces can not be declared as private because
  + private means “only visible within the enclosing class”.
  + protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

* Java

|  |
| --- |
| // Java program to illustrate error while  // Using class from different package with    // Private Modifier  **package** p1;    // Class A  **class** A {  **private** **void** display()      {          System.out.println("GeeksforGeeks");      }  }    // Class B  **class** B {  **public** **static** **void** main(String args[])      {          A obj = **new** A();          // Trying to access private method          // of another class          obj.display();      }  } |

**Output:**

error: display() has private access in A  
 obj.display();

**3. Protected Access Modifier**

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

**Program 1:**

* Java

|  |
| --- |
| // Java Program to Illustrate  // Protected Modifier  **package** p1;    // Class A  **public** **class** A {  **protected** **void** display()      {          System.out.println("GeeksforGeeks");      }  } |

**Program 2:**

* Java

|  |
| --- |
| // Java program to illustrate  // protected modifier  **package** p2;    // importing all classes in package p1  **import** p1.\*;    // Class B is subclass of A  **class** B **extends** A {  **public** **static** **void** main(String args[])      {          B obj = **new** B();          obj.display();      }  } |

**Output:**

GeeksforGeeks

**Public Access modifier**

The public access modifier is specified using the keyword **public**.

* The public access modifier has the **widest scope** among all other access modifiers.
* Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

**Program 1:**

* Java

|  |
| --- |
| // Java program to illustrate  // public modifier  **package** p1;  **public** **class** A  {  **public** **void** display()      {          System.out.println("GeeksforGeeks");      }  } |

**Program 2:**

* Java

|  |
| --- |
| **package** p2;  **import** p1.\*;  **class** B {  **public** **static** **void** main(String args[])      {          A obj = **new** A();          obj.display();      }  } |

**Output:**

GeeksforGeeks

**Important Points:**

* If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
* Avoid public fields except for constants.

**Algorithm to use access modifier in Java**

***Here’s a basic algorithm for using access modifiers in Java:***

* ***Define a class:****Create a class that represents the object you want to manage.*
* ***Define instance variables:****Within the class, define instance variables that represent the data you want to manage.*
* *Specify an access modifier: For each instance variable, specify an access modifier that determines the visibility of the variable. The three main access modifiers in Java are private, protected, and public.*
* ***Use private for variables that should only be accessible within the class:****If you want to prevent access to a variable from outside the class, use the private access modifier. This is the most restrictive access modifier and provides the greatest level of encapsulation.*
* ***Use protected for variables that should be accessible within the class and its subclasses****: If you want to allow access to a variable from within the class and its subclasses, use the protected access modifier. This is less restrictive than private and provides some level of inheritance.*
* ***Use public for variables that should be accessible from anywhere****: If you want to allow access to a variable from anywhere, use the public access modifier. This is the least restrictive access modifier and provides the least amount of encapsulation.*
* ***Use accessor and mutator methods to manage access to the variables:****In order to access and modify the variables, use accessor (getter) and mutator (setter) methods, even if the variables have a public access modifier. This provides a level of abstraction and makes your code more maintainable and testable.*

### Importance of Access Control in Java

Access control in Java is crucial for several reasons:

* **Encapsulation:** Access control is a key aspect of encapsulation in OOP. It allows the internal representation of an object to be hidden from the outside world, and only a prescribed set of methods and variables can be accessed.
* **Data Integrity:** By restricting access, you can prevent the data from being modified in unexpected ways. This helps to maintain the integrity of your data.
* **Security:** Access control can prevent sensitive data from being exposed. For instance, a private variable in a class can only be accessed within that class, preventing unauthorized access.
* **Maintainability:** Access control makes code easier to maintain. By limiting the parts of the code that can access certain components, you reduce the risk of introducing bugs when modifying the code.

Understanding the concept of access control and its importance in Java is crucial for effective and secure programming. It allows you to structure your code in a way that maximizes security, maintainability, and data integrity.

## Access Modifiers in Larger Java Projects

Access modifiers play an integral role in larger Java projects. They help maintain a clean and manageable codebase and ensure that each part of the code interacts with the others in a controlled and predictable way.

For instance, by making a variable private, you’re ensuring that it can only be accessed within its own class. This reduces the risk of it being accidentally modified from elsewhere in the code, which can be particularly valuable in a large project with multiple developers.

## Access Modifiers and Design Principles

Access modifiers in Java are also closely tied to key design principles like encapsulation and abstraction.

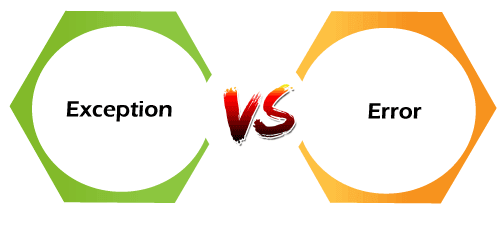
**Encapsulation** is the concept of bundling the data and methods that operate on the data within one unit and hiding the values or state of an object inside the class. It’s achieved in Java using access modifiers.

**Abstraction**, on the other hand, is the concept of hiding the complexity of code by providing a simple interface. In Java, public and protected access modifiers are often used to expose the necessary methods while keeping the implementation details hidden.

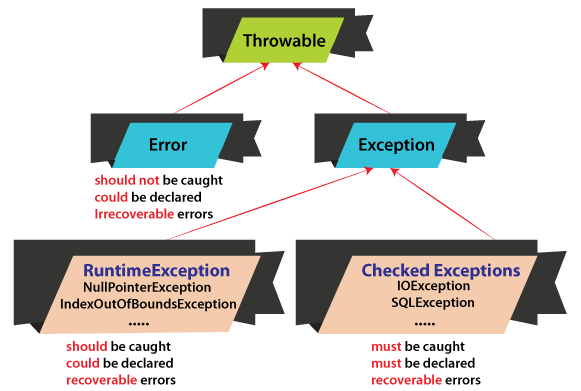
**Q.No:2**

# Exception Vs Error in Java

The general meaning of exception is **a deliberate act of omission** while the meaning of error is **an action that is inaccurate or incorrect**. In Java, **Exception,** and **Error** both are subclasses of the [Java](https://www.javatpoint.com/java-tutorial) **Throwable** class that belongs to java.lang package. But there exist some significant differences between them. So, in this section, we are going to discuss the **key differences between exception and error**.



Before moving ahead in this section let's have a look at the hierarchy of the [Java Throwable class](https://www.javatpoint.com/post/java-throwable).



## Exception

The term [**exception**](https://www.javatpoint.com/exception-handling-in-java) is shorthand for the phrase **exception event**. It is an event that occurs during the execution of the program and interrupts the normal flow of program instructions. These are the errors that occur at compile time and run time. It occurs in the code written by the developers. It can be recovered by using the try-catch block and throws keyword. There are two types of exceptions i.e. **checked** and **unchecked**.

There are some important points that should be kept in mind while dealing with the exception:

* When an error is detected, an exception is thrown.
* Any exception that is thrown must be caught by the exception handler.
* If the programmer has forgotten to provide an exception handler, the exception will be caught by the catch-all exception handler provided by the system.
* Exception may be rethrown if exception handler is failure to handle it.

### Advantages of Exceptions

* It separates error handling code from regular code.
* It has the ability to propagate error reporting up the call stack of methods.
* The grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

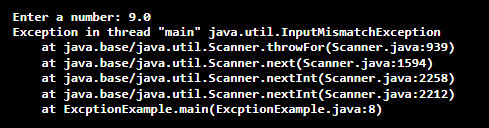
Let's understand the exception through a Java program.

### Example of Exception

**ExceptionExample.java**

1. **import** java.util.Scanner;
2. **public** **class** ExcptionExample
3. {
4. **public** **static** **void** main(String args[])
5. {
6. Scanner sc = **new** Scanner(System.in);
7. System.out.print("Enter a number: ");
8. **int** number = sc.nextInt();
9. System.out.println("You have entered: "+number);
10. }
11. }

Let's run the above program and enter a float value deliberately to generate an exception.



It shows the **InputMismatchExaception.** Because the program accepts an integer value. We observe that the next statement is skipped and the program is terminated.

## Error

[Errors](https://www.javatpoint.com/java-error) are problems that mainly occur due to the lack of system resources. It cannot be caught or handled. It indicates a serious problem. It occurs at run time. These are always unchecked. An example of errors is **OutOfMemoryError, LinkageError, AssertionError**, etc. are the subclasses of the Error class.

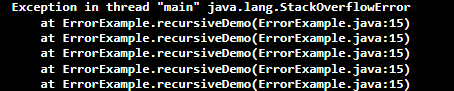
Let's understand the error through a Java program.

### Example of Error

**ErrorExample.java**

1. **public** **class** ErrorExample
2. {
3. **public** **static** **void** main(String args[])
4. {
5. //method calling
6. recursiveDemo(10);
7. }
8. **public** **static** **void** recursiveDemo(**int** i)
9. {
10. **while**(i!=0)
11. {
12. //increments the variable i by 1
13. i=i+1;
14. //recursive called method
15. recursiveDemo(i);
16. }
17. }
18. }

**Output:**



We observe that on running the program, we get the **StackOverflowError,** not an exception.

Let's discuss the key differences between exception and error.

## Difference Between Exception and Error

In Java, Error, and Exception both are subclasses of the Java Throwable class that belongs to java.lang package.

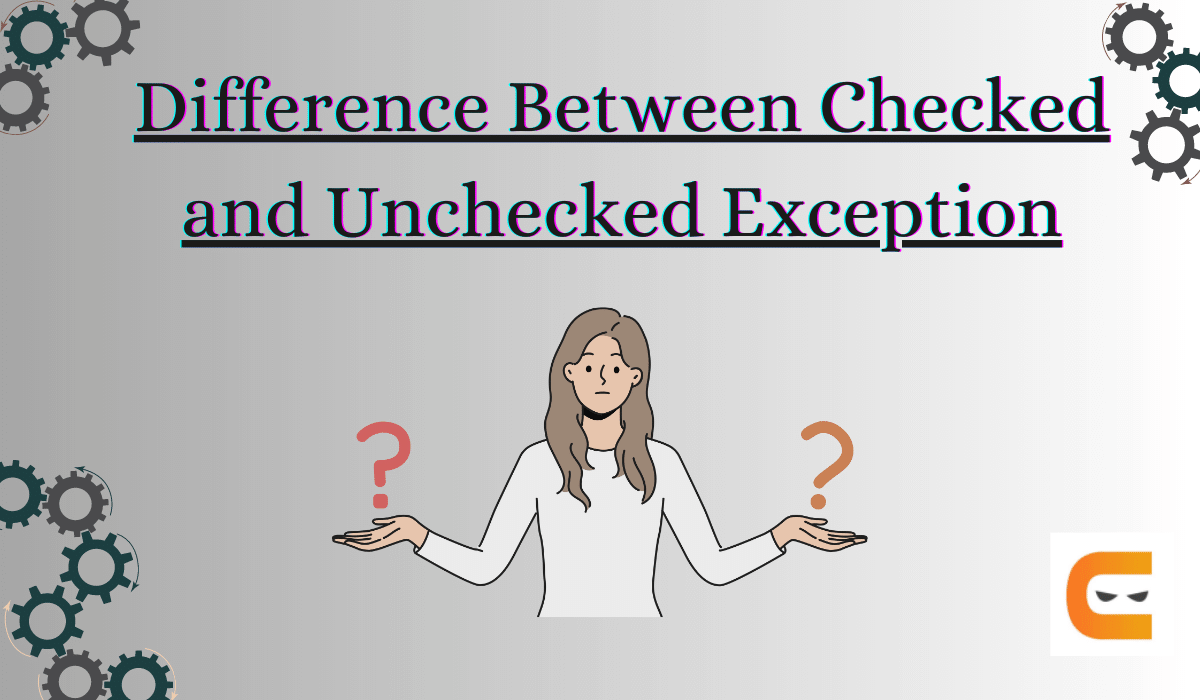
|  |  |  |
| --- | --- | --- |
| **Basis of Comparison** | **Exception** | **Error** |
| **Recoverable/ Irrecoverable** | Exception can be recovered by using the try-catch block. An error cannot be recovered. |  |
| **Type** | It can be classified into two categories i.e. checked and unchecked. | All errors in Java are unchecked. |
| **Occurrence** | It occurs at compile time or run time. | It occurs at run time. |
| **Package** | It belongs to java.lang.Exception package. | It belongs to java.lang.Error package. |
| **Known or unknown** | Only checked exceptions are known to the compiler. | Errors will not be known to the compiler. |
| **Causes** | It is mainly caused by the application itself. | It is mostly caused by the environment in which the application is running. |
| **Example** | **Checked Exceptions:** SQLException, IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundException, NullPointerException, ArithmaticException | Java.lang.StackOverFlow, java.lang.OutOfMemoryError |

**Q.No:3**

## Difference Between Checked and Unchecked Exceptions in Java

Java exception is an unexpected issue that arises while a program is running, causing it to deviate from its normal operation. These exceptions can disrupt the flow of the program and need to be handled properly to prevent crashes and ensure smooth execution. There are two types of exceptions:

1. Checked exceptions
2. Unchecked exceptions



## ****What is a Checked Exception?****

* A checked exception is the one that the compiler checks or notifies during compilation.
* Checked Exceptions are also known as **compile-time exceptions**.
* These exceptions cannot be ignored.
* If a code within a function throws a checked exception, the function must handle the exception or specify it using the *throws* keyword.

### ****Example****

A *FileReader class* is used in a file handling programme to read data from a file. If the file specified in its constructor does not exist, a **FileNotFoundException** occurs, and the compiler prompts the programmer to handle the exception.

Here, If the *Name.txt file* is not present in the specified location during the compilation, this code will throw an error.

import java.io.File;

import java.io.FileReader;

public class Test{

public static void main(String args[]) {

File file = new File("E://Name.txt");

FileReader fr = new FileReader(file);

}

}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Unhandled exception type FileNotFoundException

at Test.main(Test.java:8)

## ****hat is an Unchecked Exception?****

* An unchecked exception occurs during the execution process.
* Unchecked Exceptions are also known as **runtime exceptions**.
* Programming mistakes, such as logic errors or improper use of an API, are examples of this.
* Runtime exceptions are ignored during compilation.

### ****Example****

If a 7-element array is declared and the programme attempts to access its 8th element, an **ArrayIndexOutOfBoundsExceptionexception** occurs.

Here, the below program will be successfully compiled, but it’ll throw an exception at runtime.

public class Test{

public static void main(String args[]) {

int num[] = {100, 20, 30, 40, 50, 60, 70};

System.out.println(num[7]);

}

}

Output:

Exception in thread "main" java. Lang.ArrayIndexOutOfBoundsException: Index 8 out of bounds for length 7

at Test.main(Test.java:7)

## ****Differences between Checked and Unchecked Exceptions in Java****

| **Key Factors** | **Checked Exceptions** | **Unchecked Exceptions** |
| --- | --- | --- |
| **Occurrence** | Checked exceptions occur at compile time. | Unchecked exceptions occur at runtime. |
| **Role of Compiler** | The compiler checks a checked exception at compile time. | The compiler does not check these exceptions. |
| **Handling** | Handled at compile time. | Handled at run time. |
| **Parent Class** | Exception class is the direct parent class of the checked exception subclass. | RuntimeException is the direct parent class of the Unchecked exception subclass. |
| **Reason for Occurrence** | A checked Exception occurs when the chances of failure are too high. | RuntimeException is the direct parent class of the Unchecked exception subclass. |
| **Example** | SQLException, IOException, ClassNotFoundException, InvocationTargetException | NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, IllegalArgumentException,NumberFormatException |