1. Exception Handling in Java

I. Introduction

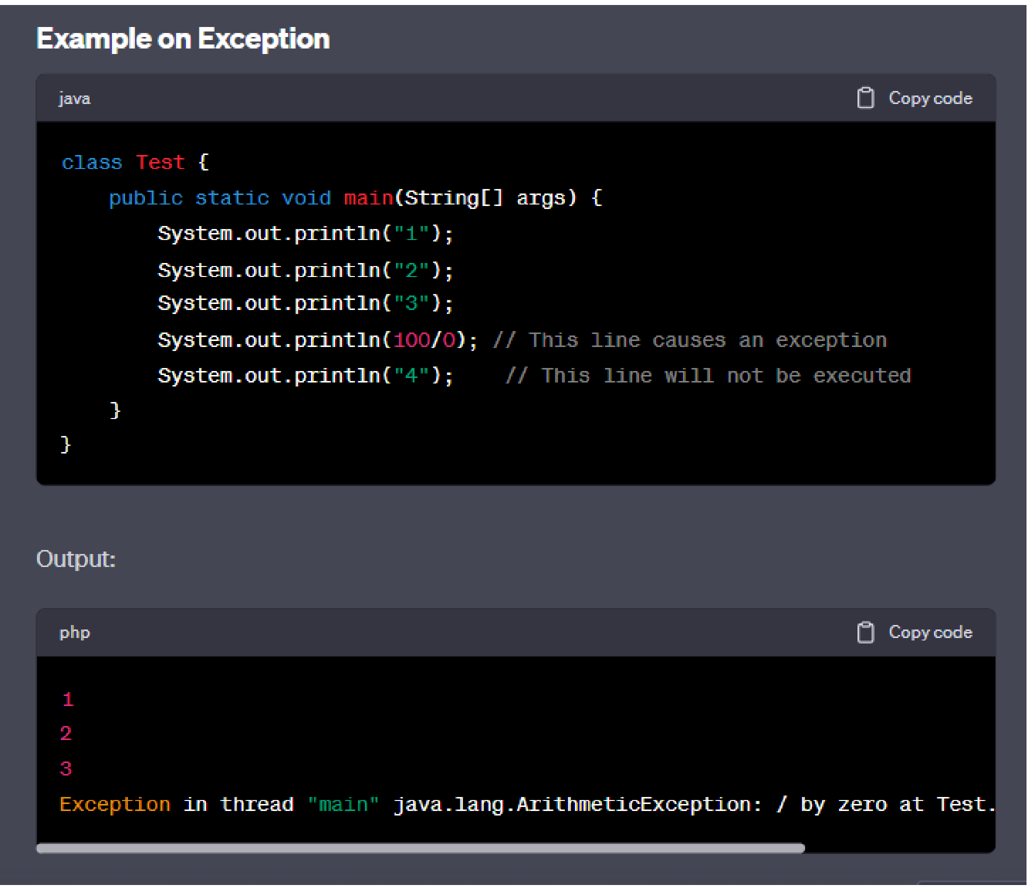
- An Exception is an unwanted or unexpected event that occurs during the execution of a program.
- At runtime, it disrupts the normal flow of the program's instructions.
- When an exception occurs, the method in which the exception occurs will create an **EXCEPTION OBJECT** that stores three things:

  1. **Exception name**
  2. **Description**
  3. **Stack trace**

II. Exception Handling

- Exception handling is a mechanism for managing exceptions or using alternative methods to continue the execution flow without disturbances.
- Consider the following analogy: Imagine having to reach the institute daily on time using a bike. This becomes the normal flow.
- However, one day, the bike's tire gets punctured, and you can't reach the institute on time.
- This disruption is an exception. To handle it, you'd need to find an alternative way to reach the institute. This is the essence of exception handling.

III. Example on Exception



Example on Exception

java Copy code

```java
class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("2");
        System.out.println("3");
        System.out.println(100/0); // This line causes an exception
        System.out.println("4");    // This line will not be executed
    }
}
```

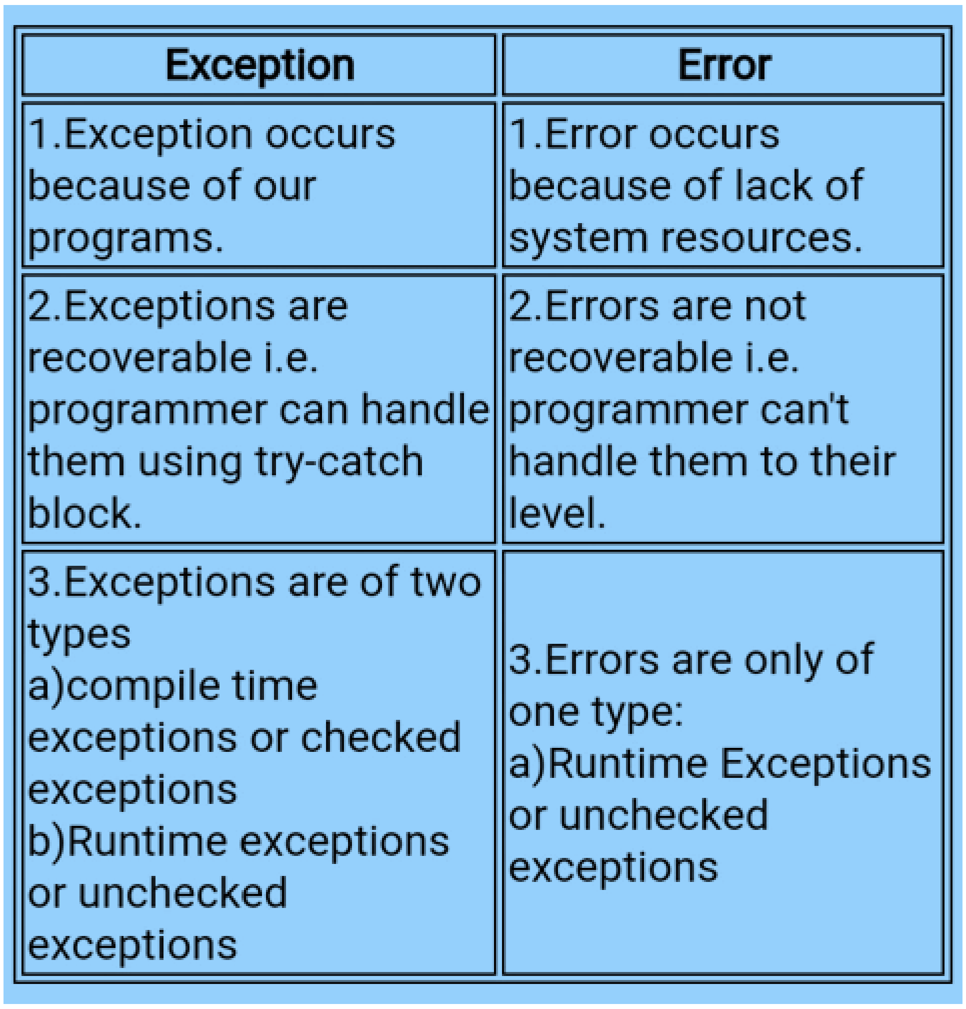
Output:

php Copy code

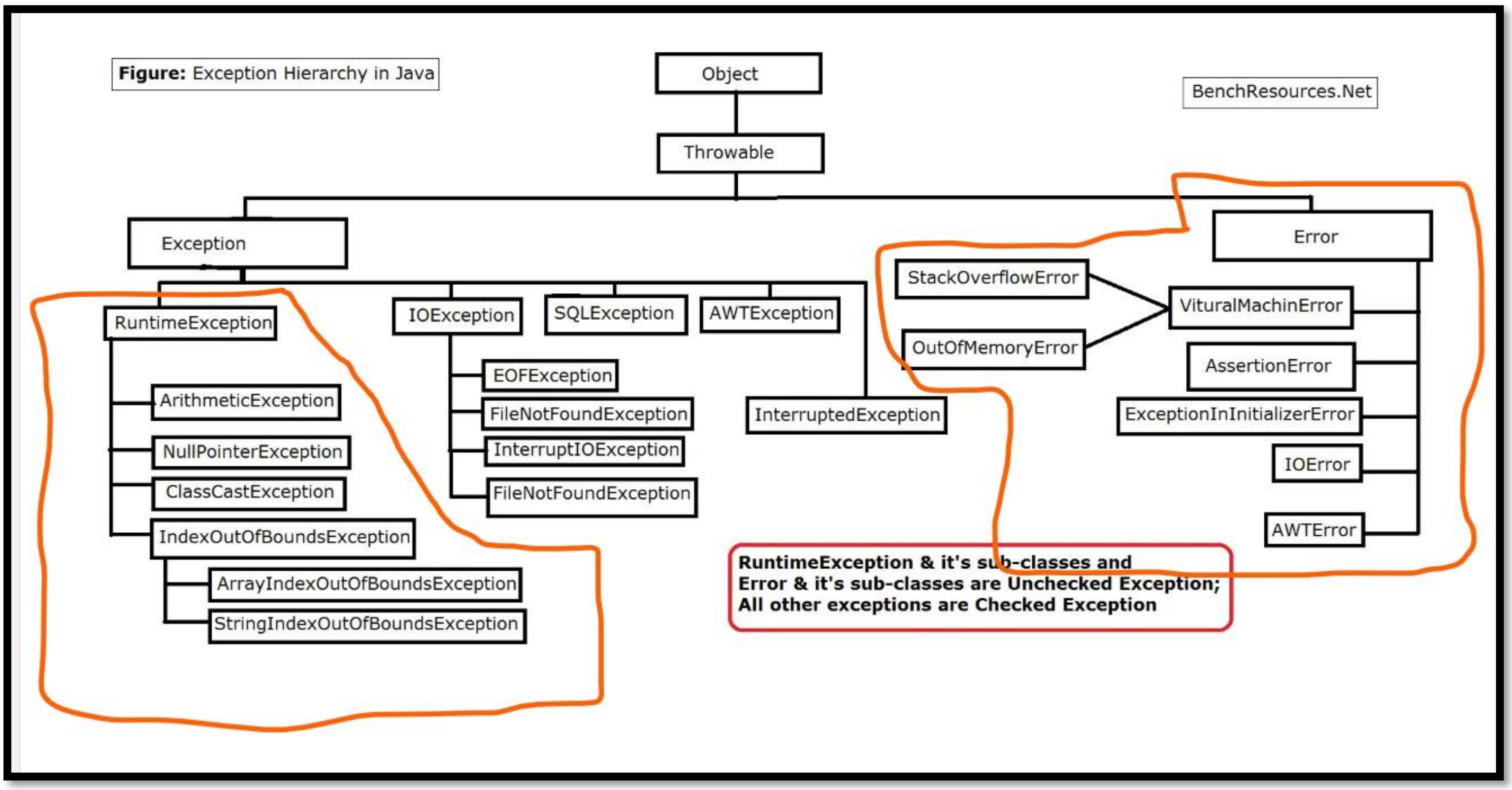
```
1
2
3
Exception in thread "main" java.lang.ArithmeticException: / by zero at Test.
```

In this example, after line 8, the subsequent statements are not executed due to the exception. These types of exceptions need proper mechanisms to handle to prevent software crashes.

2. Difference between Exception and Error



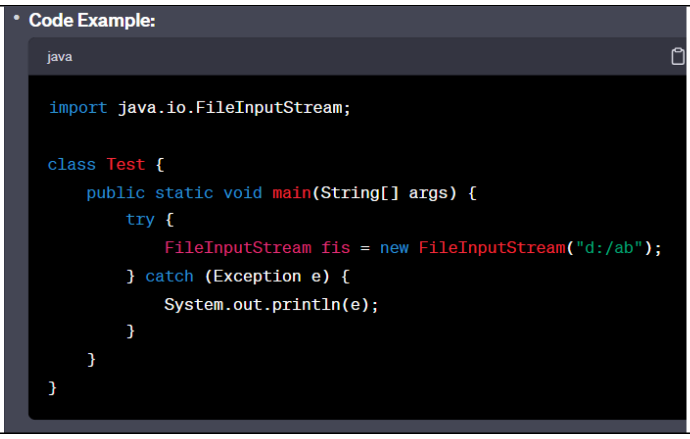
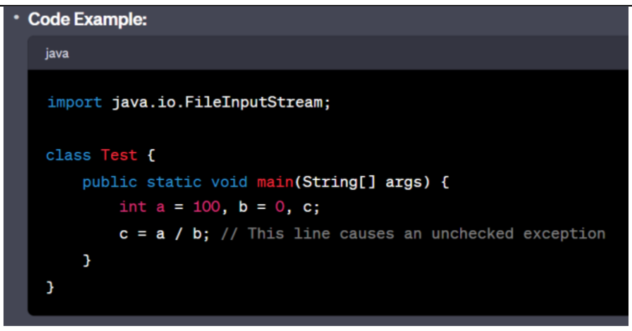
Exception	Error
1.Exception occurs because of our programs.	1.Error occurs because of lack of system resources.
2.Exceptions are recoverable i.e. programmer can handle them using try-catch block.	2.Errors are not recoverable i.e. programmer can't handle them to their level.
3.Exceptions are of two types a)compile time exceptions or checked exceptions b)Runtime exceptions or unchecked exceptions	3.Errors are only of one type: a)Runtime Exceptions or unchecked exceptions



RuntimeException & it's sub-classes and Error & it's sub-classes are Unchecked Exception; All other exceptions are Checked Exception

3. Difference between Checked and Unchecked Exceptions

- In Java, exceptions play a crucial role in handling errors and exceptional situations.
- There are two main categories of exceptions: Checked and Unchecked.

Aspect	Compile-time Exceptions (Checked Exception)	Run-time Exceptions (Unchecked Exception)
Definition	Exceptions that the compiler checks at compile time.	Exceptions not checked by the compiler at compile time.
Handling Approach	Must be either handled using try-catch blocks or declared with **throws**.	Typically handled using runtime checks and exception handling.
Handling Flexibility	Anticipated exceptions, can be explicitly handled or propagated.	Often arise due to logical errors, may be harder to anticipate.
Example Analogy	Like your mother checking your purse or ID card before work.	Similar to attempting to divide by zero, leading to unexpected behavior.
Example CODE		

- **Code Example:** (Checked Exception)

java
```java
import java.io.FileInputStream;

class Test {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("d:/ab");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

- **Code Example:** (Unchecked Exception)

java
```java
import java.io.FileInputStream;

class Test {
    public static void main(String[] args) {
        int a = 100, b = 0, c;
        c = a / b; // This line causes an unchecked exception
    }
}
```

- In summary, the key distinction between checked and unchecked exceptions lies in the compiler's ability to anticipate and verify them at compile time.
- While checked exceptions must be handled or declared, unchecked exceptions may occur due to logical errors during runtime.

4. Try and Catch in Java

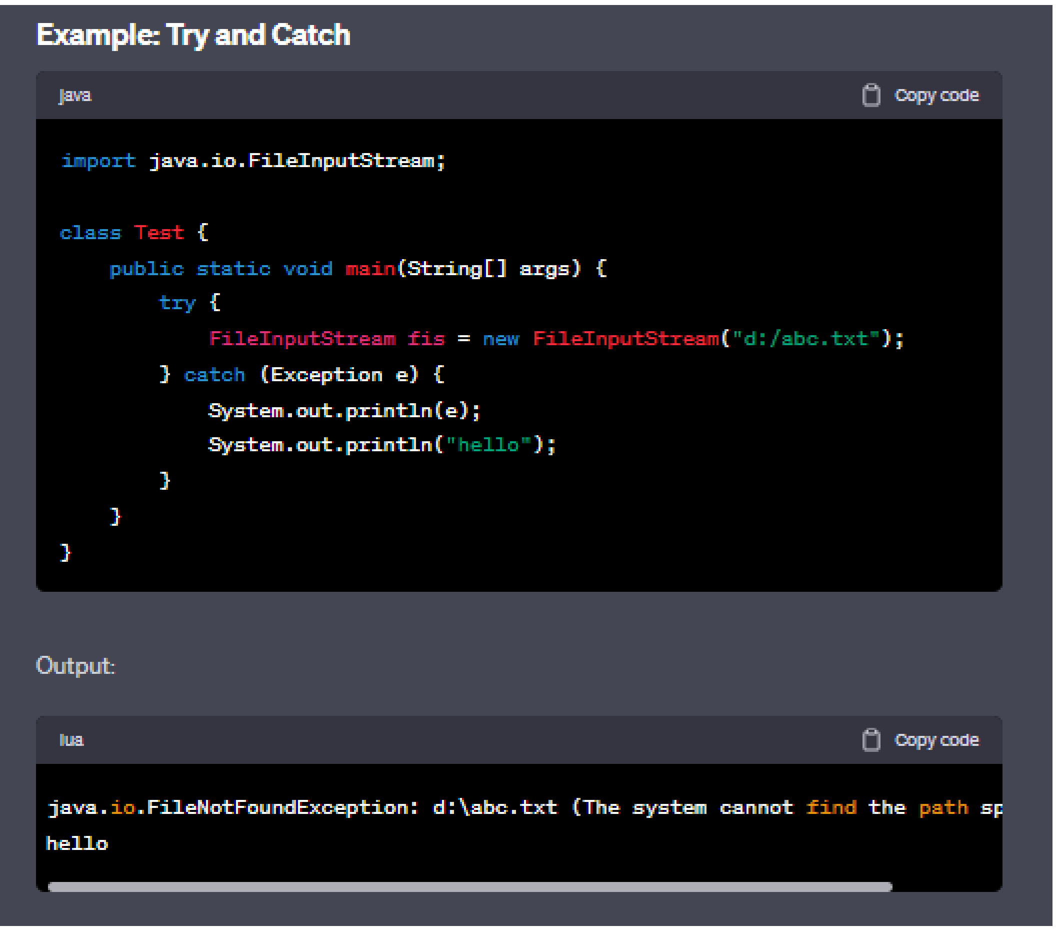
I. *Definition*

Java exception handling is managed using five keywords:

1. **try**
2. **catch**
3. **finally**
4. **throw**
5. **throws**

II. *Try and Catch Keywords*

1) **try** is a block that contains code that might throw an exception.
2) The **try** block contains risky code that can cause errors in the program.
3) The **catch** block contains code for handling exceptions.
4) The **catch** clause takes the exception as a parameter.
5) **try** and **catch** blocks work in pairs.
6) Multiple catch blocks can be used at the same time.



Example: Try and Catch

```java
import java.io.FileInputStream;

class Test {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("d:/abc.txt");
        } catch (Exception e) {
            System.out.println(e);
            System.out.println("hello");
        }
    }
}
```

Output:

```
java.io.FileNotFoundException: d:\abc.txt (The system cannot find the path sp
hello
```

III. *Control Flow of Try and Catch*

1. When using try and catch, the program flows from top to bottom.
2. If an exception occurs in the try block, the remaining statements in the try block are not executed.
3. If there is no exception in the program, the statements in the catch block will not be executed.

5. Printing Exception Information in Java

In Java, there are three ways to print exception information when an exception occurs:

a. *Using **printStackTrace()** method:*

Displays detailed information about the exception, including the exception type and the line number where the exception occurred.

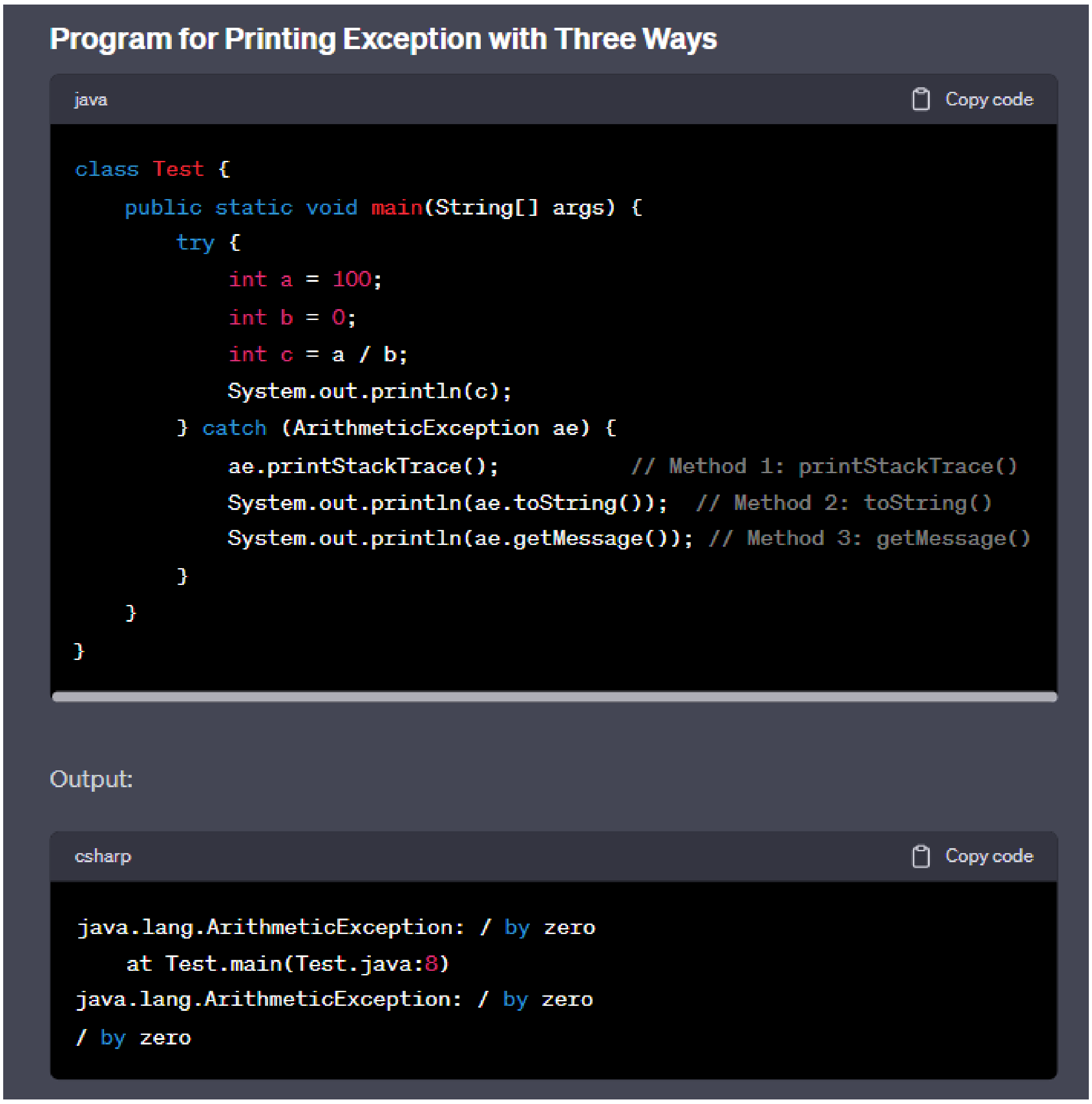
b. *Using **toString()** method:*

Returns a string representation of the exception.

c. *Using **getMessage()** method:*

Retrieves the error message associated with the exception.

I. *Program for Printing Exception with Three Ways*



Program for Printing Exception with Three Ways

```java
class Test {
    public static void main(String[] args) {
        try {
            int a = 100;
            int b = 0;
            int c = a / b;
            System.out.println(c);
        } catch (ArithmeticException ae) {
            ae.printStackTrace();           // Method 1: printStackTrace()
            System.out.println(ae.toString());  // Method 2: toString()
            System.out.println(ae.getMessage()); // Method 3: getMessage()
        }
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
    at Test.main(Test.java:8)
java.lang.ArithmeticException: / by zero
/ by zero
```

In this program, an **ArithmeticException** is intentionally triggered by dividing a number by zero. The program then demonstrates three different methods to print exception information:

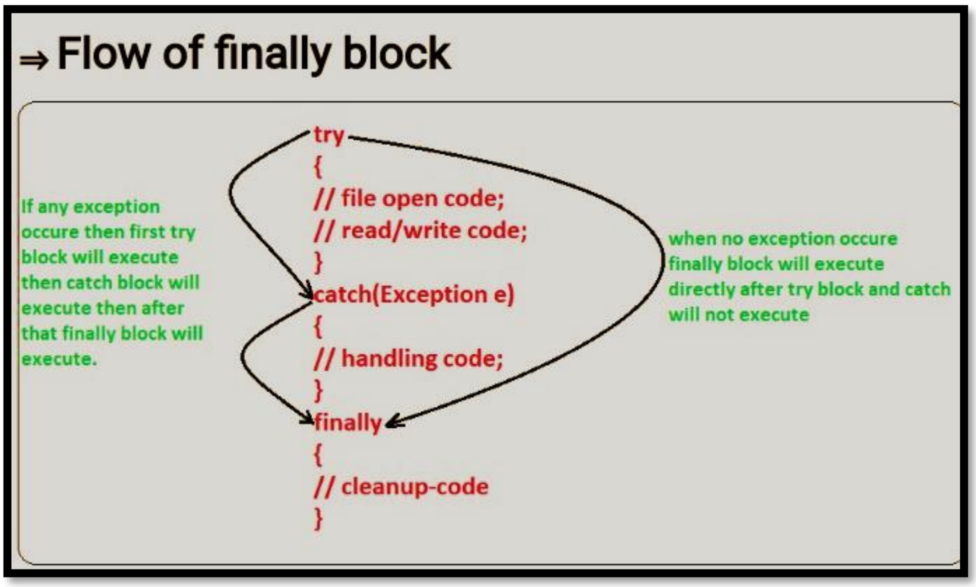
6. Finally Block in Java

The **finally** block is a crucial component in Java exception handling. It is typically used in conjunction with the **try** and **catch** blocks. The code within the **finally** block is guaranteed to execute, regardless of whether an exception occurs or not. This block often contains important cleanup code, such as closing connections or streams.

I. *Flow of the finally Block*

In the flow of execution:

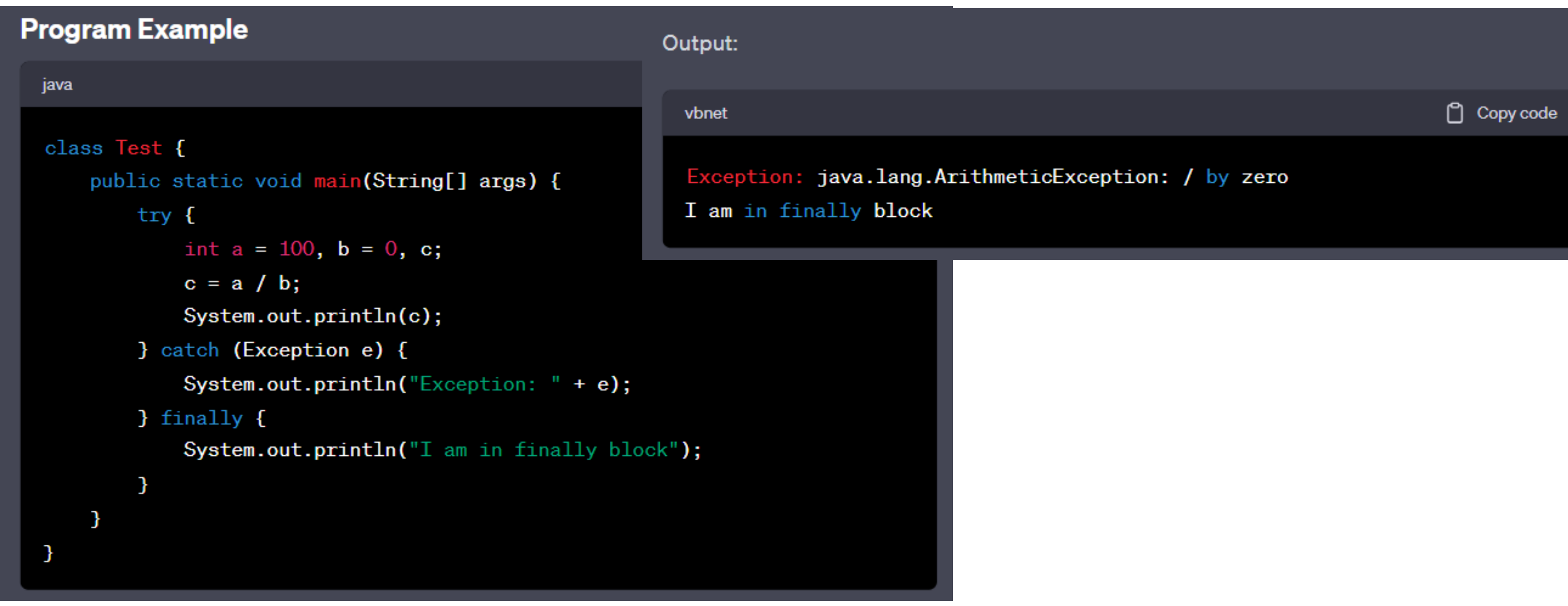
1) If an exception occurs, the **try** block is executed.
2) If an exception is caught, the corresponding **catch** block is executed.
3) Regardless of whether an exception occurred or was caught, the code in the **finally** block will execute.


⇒ Flow of finally block
If any exception occure then first try block will execute then catch block will execute then after that finally block will execute.
try
{
// file open code;
// read/write code;
}
catch(Exception e)
{
// handling code;
}
finally
{
// cleanup-code
}
when no exception occure finally block will execute directly after try block and catch will not execute

II. *Key Points about the finally Block*

1) You can use multiple **catch** blocks with a single **try** block, but only one **finally** block can be associated with a **try** block.
2) Statements within the **finally** block execute even if the **try** block contains control transfer statements like **return**, **break**, or **continue**.
3) However, certain actions can disturb the execution of the **finally** block:
   a. Using the **System.exit()** method.
   b. Causing a fatal error leading to process abortion.
   c. The death of a thread.
   d. An exception arising within the **finally** block itself.

III. *Program Example*



Program Example

java

```java
class Test {
    public static void main(String[] args) {
        try {
            int a = 100, b = 0, c;
            c = a / b;
            System.out.println(c);
        } catch (Exception e) {
            System.out.println("Exception: " + e);
        } finally {
            System.out.println("I am in finally block");
        }
    }
}
```

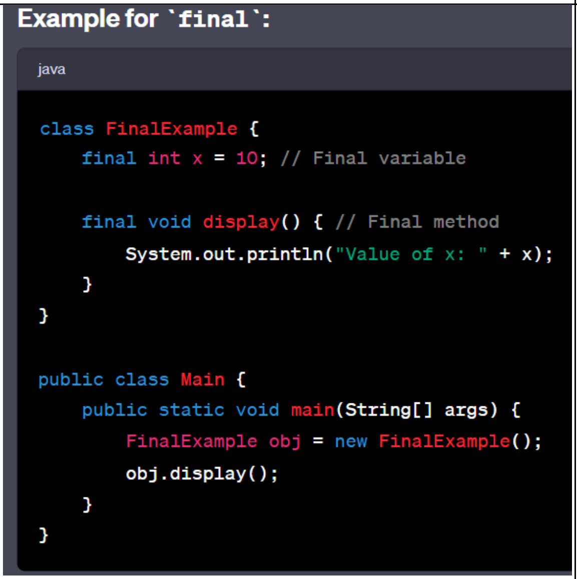
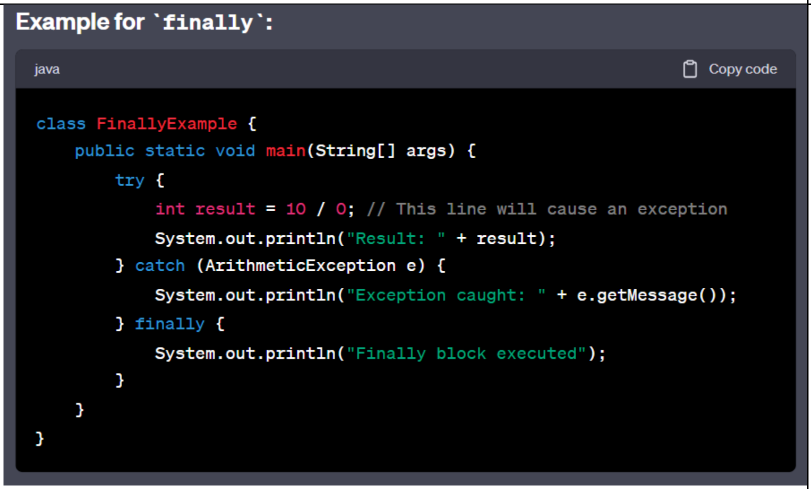
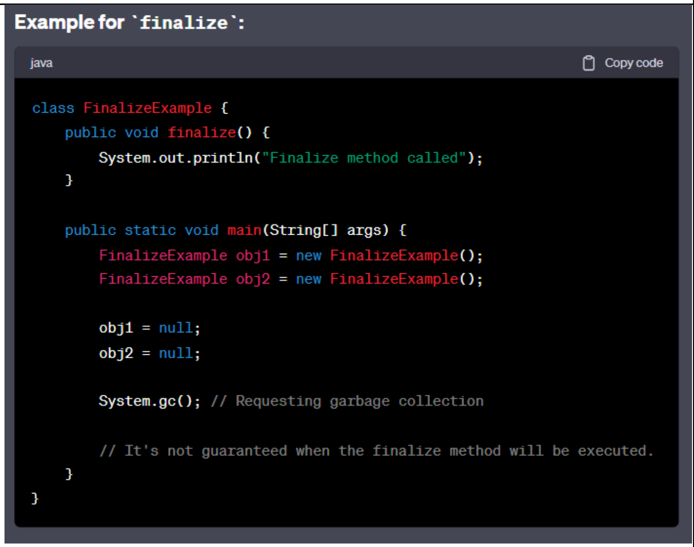
Output:

vbnet

```
Exception: java.lang.ArithmeticException: / by zero
I am in finally block
```

In this example, the **try** block throws an **ArithmeticException**, which is caught in the corresponding **catch** block. Regardless of the exception, the code within the **finally** block executes afterward.

7.Difference between final,finally and finalize

Aspect	final	finally	finalize
type	Keyword	Block	Method
Usage	Used with variables, methods, and classes.	Used with **try** or **try-catch** blocks.	Overridden for an object.
Purpose	Implies that the value of a variable cannot be changed, a method cannot be overridden, or a class cannot be subclassed.	Contains important code to be executed regardless of exceptions.	Executes just prior to garbage collection.
Garbage Collectio n	Not related to garbage collection.	Not related to garbage collection.	Executes before an object is reclaimed by garbage collection.
Execution	No specific relation to exception handling.	Executes after the **try** or **try-catch** block, whether an exception occurs or not.	Executes before an object is garbage collected.
Example			

Example for `final`:

java

```java
class FinalExample {
    final int x = 10; // Final variable

    final void display() { // Final method
        System.out.println("Value of x: " + x);
    }
}

public class Main {
    public static void main(String[] args) {
        FinalExample obj = new FinalExample();
        obj.display();
    }
}
```

Example for `finally`:

java Copy code

```java
class FinallyExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This line will cause an exception
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

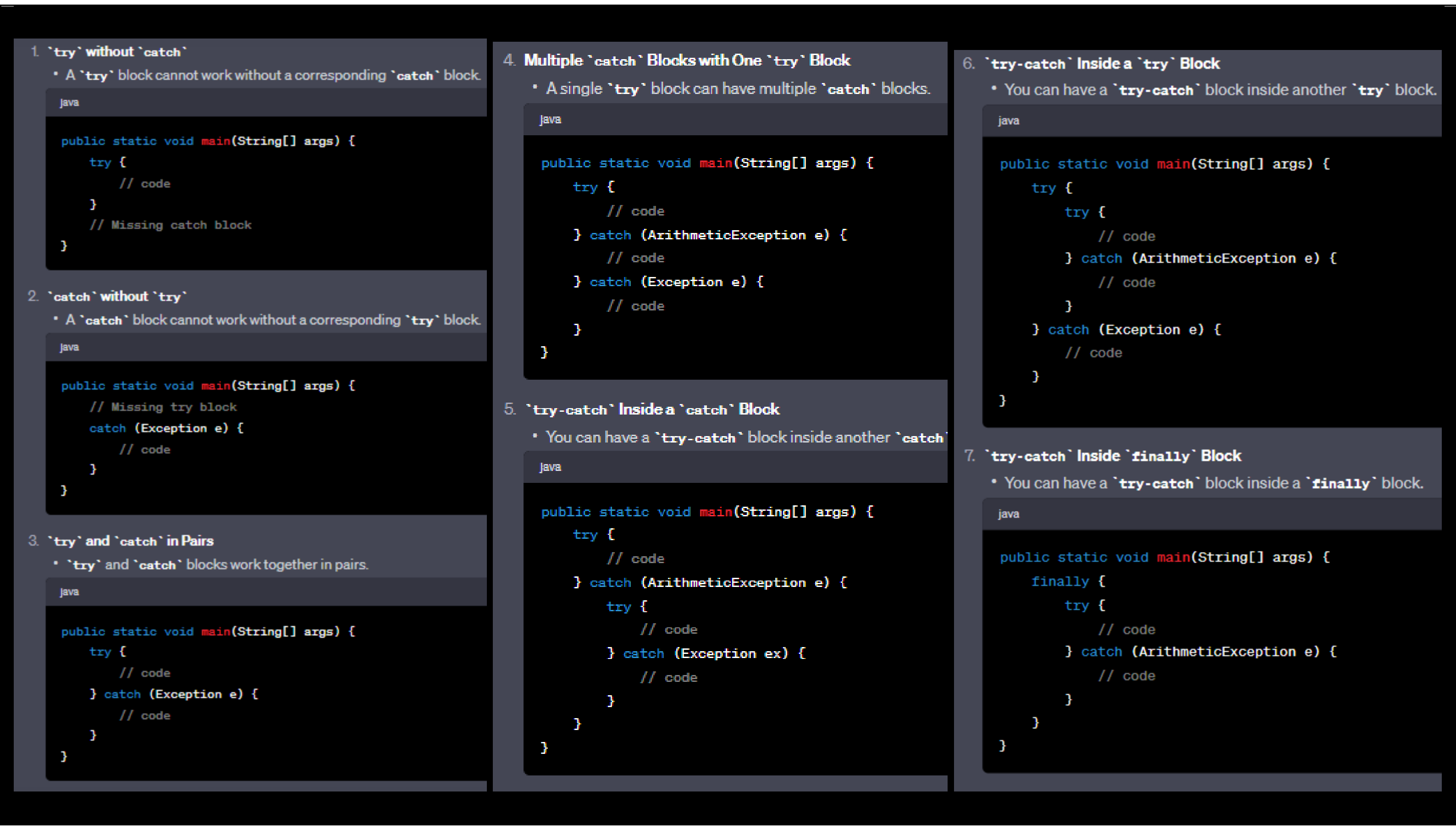
Example for `finalize`:

java Copy code

```java
class FinalizeExample {
    public void finalize() {
        System.out.println("Finalize method called");
    }

    public static void main(String[] args) {
        FinalizeExample obj1 = new FinalizeExample();
        FinalizeExample obj2 = new FinalizeExample();

        obj1 = null;
        obj2 = null;

        System.gc(); // Requesting garbage collection

        // It's not guaranteed when the finalize method will be executed.
    }
}
```

8. Possible Combinations of try, catch, and finally in Java

1. **try** without **catch**
2. **catch** without **try**
3. **try** and **catch** in Pairs
4. Multiple **catch** Blocks with One **try** Block
5. **try-catch** Inside a **catch** Block
6. **try-catch** Inside a **try** Block
7. **try-catch** Inside **finally** Block



1. **`try` without `catch`**
   - A `try` block cannot work without a corresponding `catch` block.

java
```java
public static void main(String[] args) {
    try {
        // code
    }
    // Missing catch block
}
```

2. **`catch` without `try`**
   - A `catch` block cannot work without a corresponding `try` block.

java
```java
public static void main(String[] args) {
    // Missing try block
    catch (Exception e) {
        // code
    }
}
```

3. **`try` and `catch` in Pairs**
   - `try` and `catch` blocks work together in pairs.

java
```java
public static void main(String[] args) {
    try {
        // code
    } catch (Exception e) {
        // code
    }
}
```

4. **Multiple `catch` Blocks with One `try` Block**
   - A single `try` block can have multiple `catch` blocks.

java
```java
public static void main(String[] args) {
    try {
        // code
    } catch (ArithmeticException e) {
        // code
    } catch (Exception e) {
        // code
    }
}
```

5. **`try-catch` Inside a `catch` Block**
   - You can have a `try-catch` block inside another `catch`

java
```java
public static void main(String[] args) {
    try {
        // code
    } catch (ArithmeticException e) {
        try {
            // code
        } catch (Exception ex) {
            // code
        }
    }
}
```

6. **`try-catch` Inside a `try` Block**
   - You can have a `try-catch` block inside another `try` block.

java
```java
public static void main(String[] args) {
    try {
        try {
            // code
        } catch (ArithmeticException e) {
            // code
        }
    } catch (Exception e) {
        // code
    }
}
```

7. **`try-catch` Inside `finally` Block**
   - You can have a `try-catch` block inside a `finally` block.

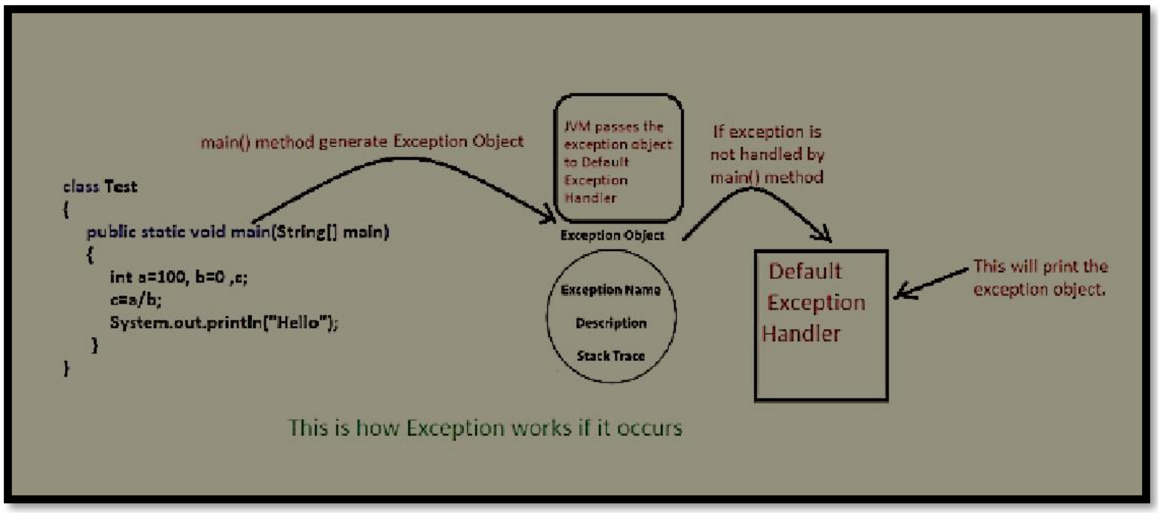
java
```java
public static void main(String[] args) {
    finally {
        try {
            // code
        } catch (ArithmeticException e) {
            // code
        }
    }
}
```

9. Throw Keyword in Java

The **throw** keyword in Java is primarily used in the context of custom exceptions. It is utilized to explicitly throw an exception, allowing programmers to generate their own exceptions. Both unchecked and checked exceptions can be thrown using the **throw** keyword. When using the **throw** keyword, the programmer takes charge of throwing the exception.

I. *Default Working of Exceptions*

1. The **main()** method generates an Exception Object.
2. The JVM passes the exception object to the Default Exception Handler.
3. If the exception is not handled by the **main()** method, the Default Exception Handler prints the exception object.



```
class Test
{
    public static void main(String[] main)
    {
        int a=100, b=0 ,c;
        c=a/b;
        System.out.println("Hello");
    }
}
```

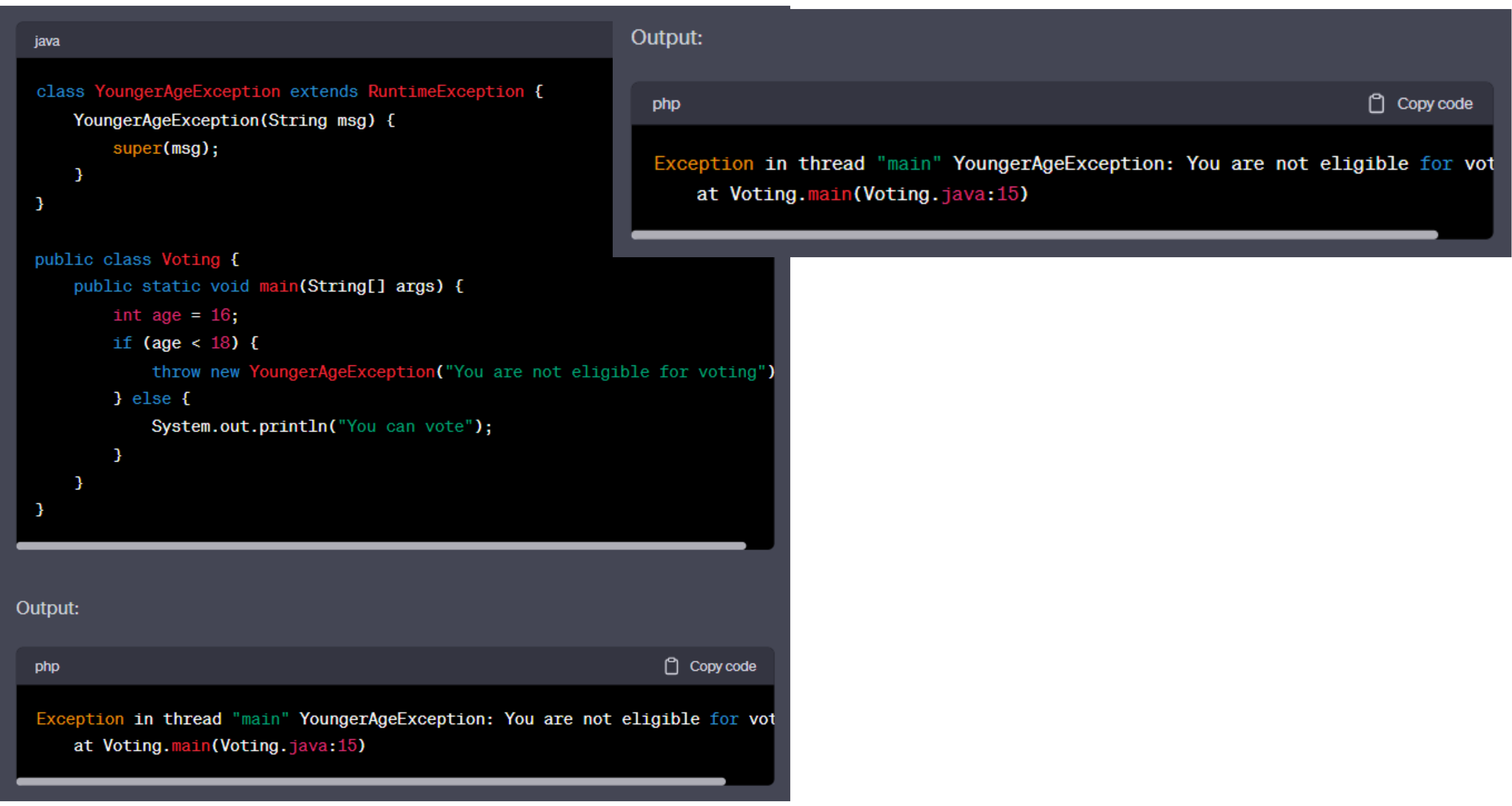
main() method generate Exception Object
JVM passes the exception object to Default Exception Handler
Exception Object
Exception Name
Description
Stack Trace
If exception is not handled by main() method
Default Exception Handler
This will print the exception object.
This is how Exception works if it occurs

II. *Throw Keyword Usage*

The **throw** keyword is employed when an exception is required in a specific scenario or according to project conditions. It allows the programmer to create an exception object and pass it to the JVM.

III. *Example: Using Throw Keyword*

Let's consider a scenario where a project involves voting, and an exception should be thrown if the voter's age is less than 18. In this case, a custom exception is needed, and the **throw** keyword is used. The programmer creates an exception object and passes it to the JVM, which is then caught by the Default Exception Handler.



java

```java
class YoungerAgeException extends RuntimeException {
    YoungerAgeException(String msg) {
        super(msg);
    }
}

public class Voting {
    public static void main(String[] args) {
        int age = 16;
        if (age < 18) {
            throw new YoungerAgeException("You are not eligible for voting")
        } else {
            System.out.println("You can vote");
        }
    }
}
```

Output:

php

```
Exception in thread "main" YoungerAgeException: You are not eligible for vot
    at Voting.main(Voting.java:15)
```

Output:

php

```
Exception in thread "main" YoungerAgeException: You are not eligible for vot
    at Voting.main(Voting.java:15)
```

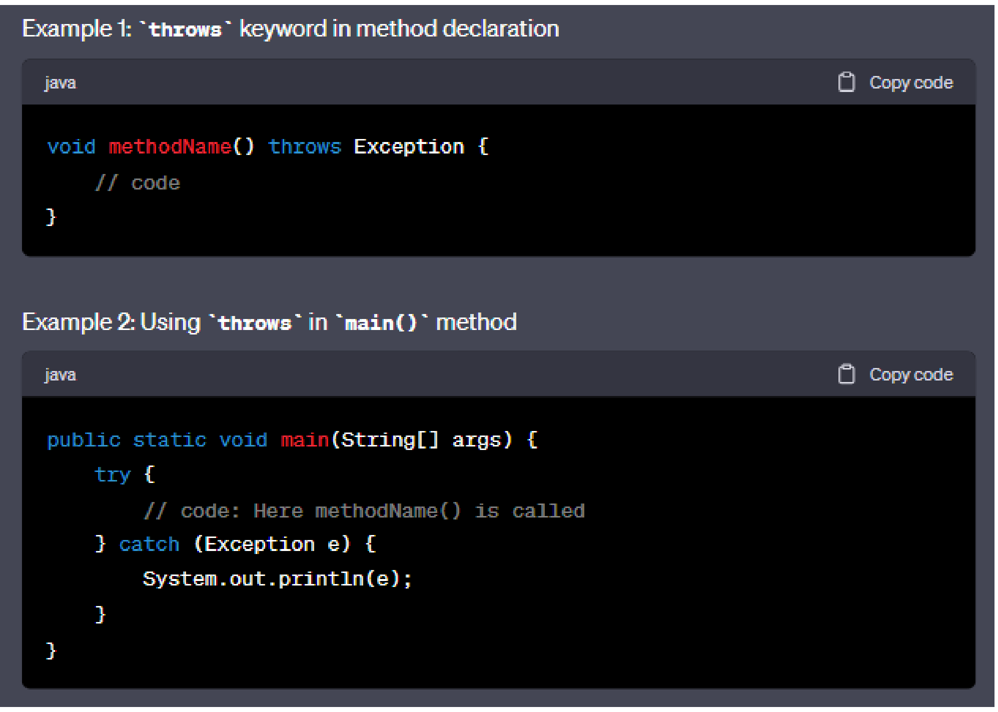
In this example, the **throw** keyword is used to create and throw a custom exception **YoungerAgeException** if the age is below 18.

10. throws Keyword in Java

- The **throws** keyword in Java is used to declare an exception.
- It provides information to the caller method that an exception might occur, allowing the caller to provide the necessary exception handling code.
- This helps in maintaining the normal flow of execution.
- If an exception occurs in a method and the programmer wants another method or programmer to handle that specific exception in their own way, the **throws** keyword is used with the method signature along with the exception name.
- **Note**: The **throws** keyword is used to declare checked exceptions. If an unchecked exception (e.g., **NullPointerException**) occurs, it indicates that the programmer hasn't performed necessary checks before using the code.

I. *Working of throws Keyword*

An exception is thrown to the caller method (e.g., **main()** method), and it handles the exception in its own way.



Example 1: `throws` keyword in method declaration

java Copy code

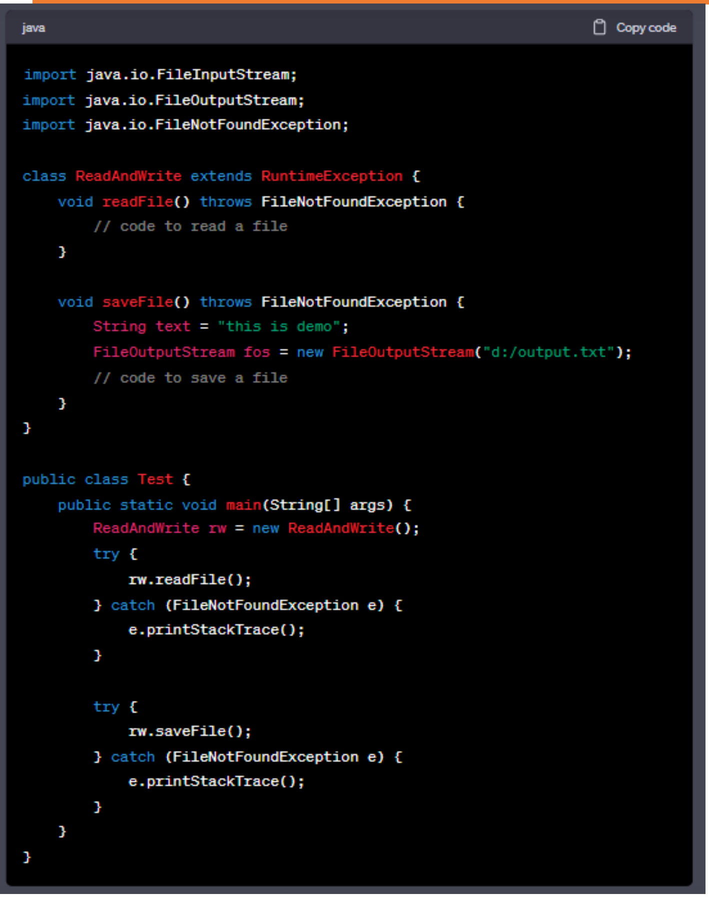
```java
void methodName() throws Exception {
    // code
}
```

Example 2: Using `throws` in `main()` method

java Copy code

```java
public static void main(String[] args) {
    try {
        // code: Here methodName() is called
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

II. *Example: Using throws Keyword*



java Copy code

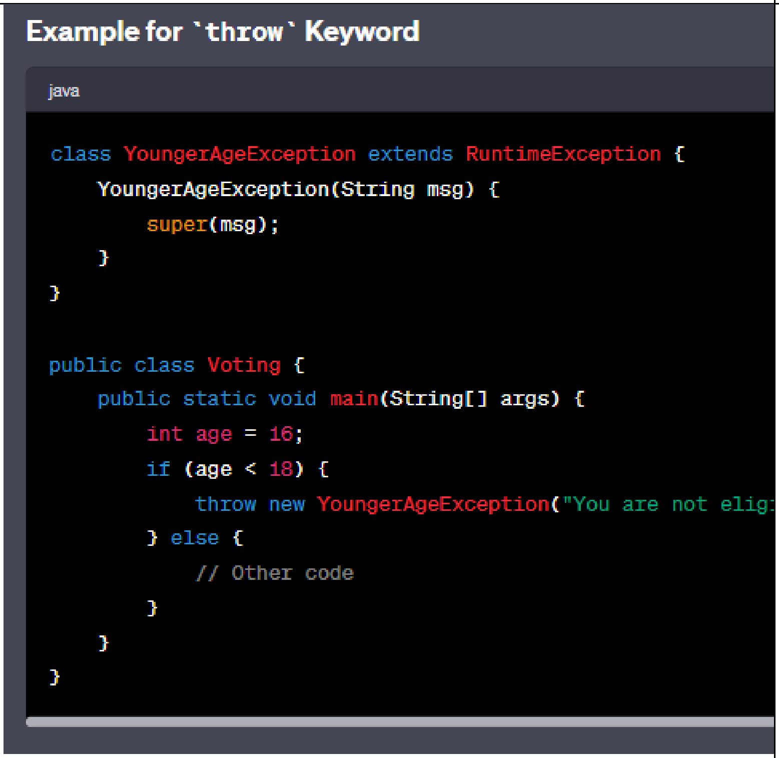
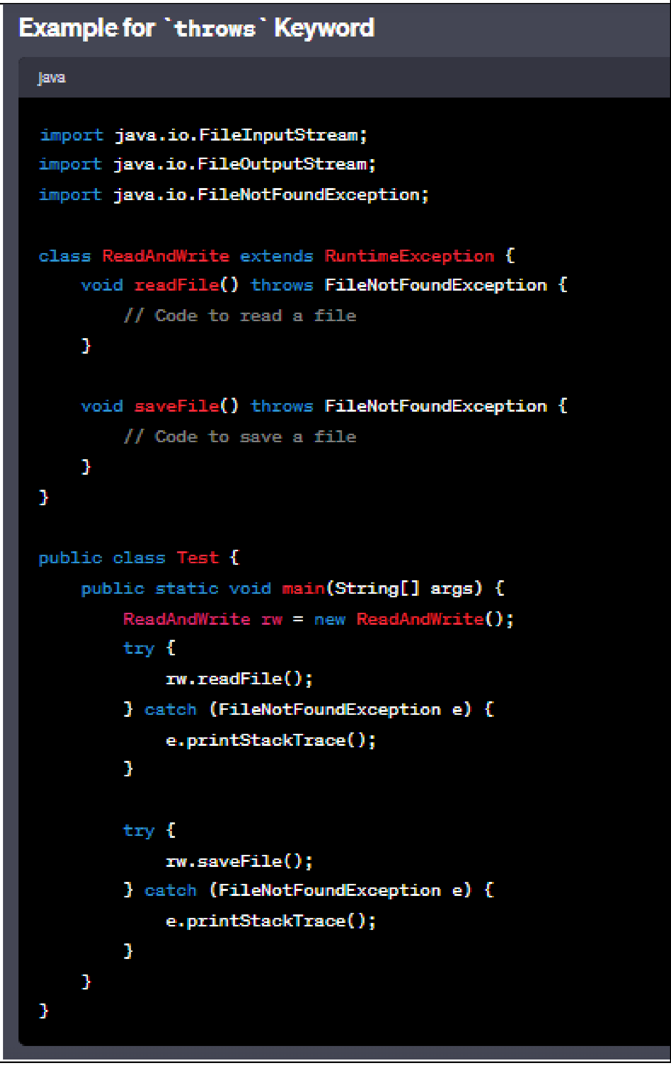
```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

class ReadAndWrite extends RuntimeException {
    void readFile() throws FileNotFoundException {
        // code to read a file
    }

    void saveFile() throws FileNotFoundException {
        String text = "this is demo";
        FileOutputStream fos = new FileOutputStream("d:/output.txt");
        // code to save a file
    }
}

public class Test {
    public static void main(String[] args) {
        ReadAndWrite rw = new ReadAndWrite();
        try {
            rw.readFile();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        try {
            rw.saveFile();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, the **throws** keyword is used to declare that the **readFile()** and **saveFile()** methods may throw a **FileNotFoundException**.

OUTPUT

```
java.io.FileNotFoundException:

d:\abc.txt (The system cannot find the path specified)
```

11. differences between the throw and throws keywords

Aspect	throw Keyword	throws Keyword
Purpose	Used to manually create an exception object.	Used to declare exceptions that a method may throw.
Exception Type	Mainly used for runtime (unchecked) exceptions.	Mainly used for compile-time (checked) exceptions.
Number of Exceptions	Only one exception can be thrown using **throw**.	Multiple exceptions can be declared using **throws**.
Usage	Inside a method.	In the method signature.
Followed by	New instance of an exception class.	List of exception classes separated by commas.
Additional Code	No other statements can follow **throw**.	No specific rule for additional statements.
Example	**throw new CustomException("Message");**	**void method() throws ExceptionType1, ExceptionType2 {...}**
		

Example for `throw` Keyword

java

```java
class YoungerAgeException extends RuntimeException {
    YoungerAgeException(String msg) {
        super(msg);
    }
}

public class Voting {
    public static void main(String[] args) {
        int age = 16;
        if (age < 18) {
            throw new YoungerAgeException("You are not elig
        } else {
            // Other code
        }
    }
}
```

Example for `throws` Keyword

java

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

class ReadAndWrite extends RuntimeException {
    void readFile() throws FileNotFoundException {
        // Code to read a file
    }

    void saveFile() throws FileNotFoundException {
        // Code to save a file
    }
}

public class Test {
    public static void main(String[] args) {
        ReadAndWrite rw = new ReadAndWrite();
        try {
            rw.readFile();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        try {
            rw.saveFile();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

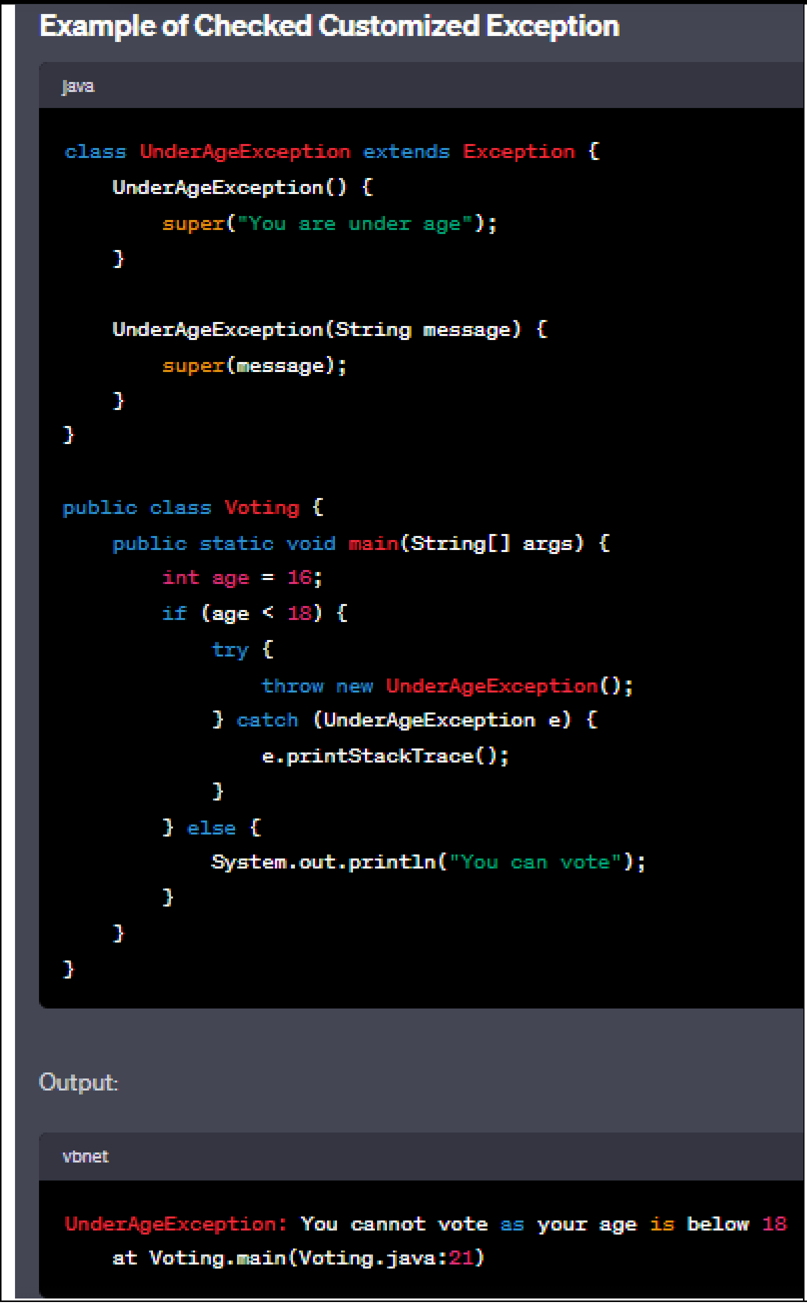
12. Customized (User-Defined) Exceptions in Java

Customized or user-defined exceptions refer to the creation of our own exceptions in Java. These exceptions are designed to meet specific user needs, allowing customization of the exception messages and behavior.

I. *Customized exceptions can be categorized into two types:*

1. Checked Customized Exception
2. Unchecked Customized Exception

Example of Checked Customized Exception	Example of Unchecked Customized Exception



Example of Checked Customized Exception

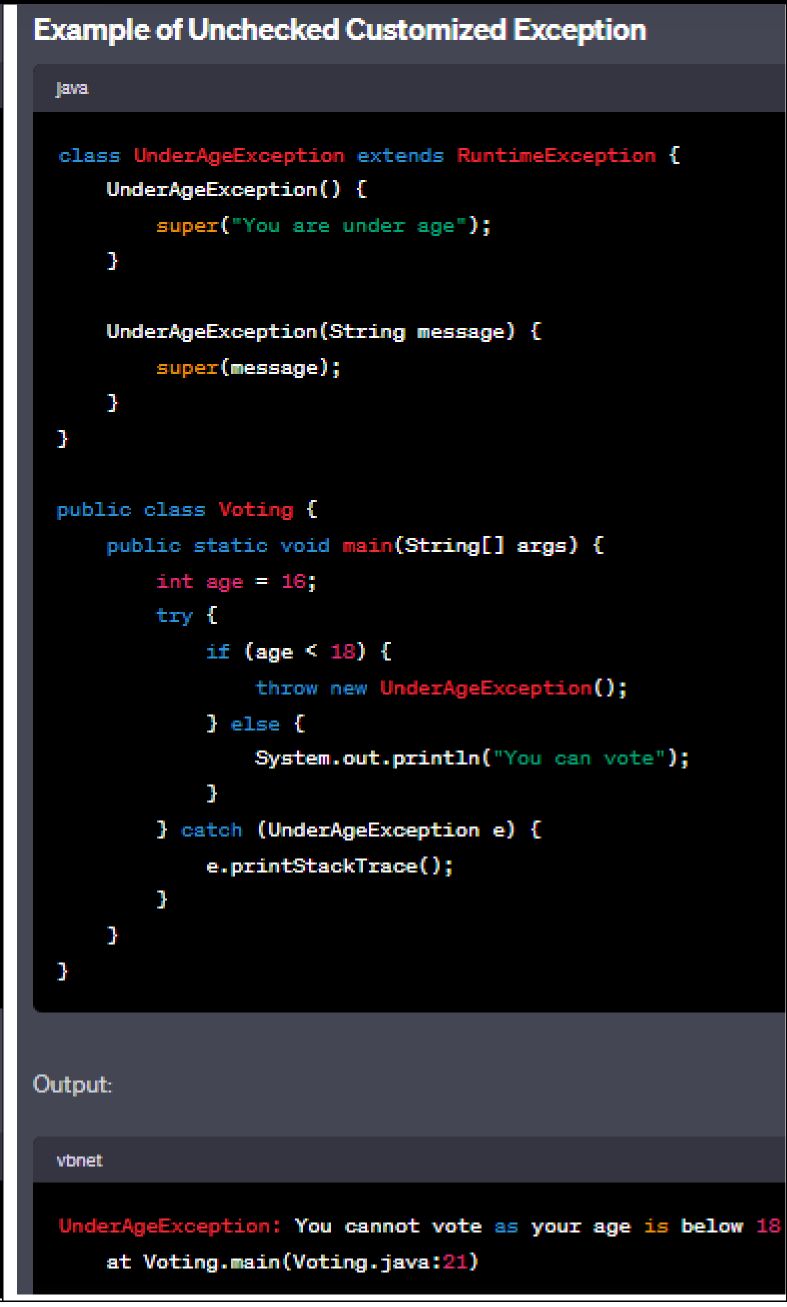
java

```java
class UnderAgeException extends Exception {
    UnderAgeException() {
        super("You are under age");
    }

    UnderAgeException(String message) {
        super(message);
    }
}

public class Voting {
    public static void main(String[] args) {
        int age = 16;
        if (age < 18) {
            try {
                throw new UnderAgeException();
            } catch (UnderAgeException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("You can vote");
        }
    }
}
```

Output:

vbnet

```
UnderAgeException: You cannot vote as your age is below 18
    at Voting.main(Voting.java:21)
```



Example of Unchecked Customized Exception

java

```java
class UnderAgeException extends RuntimeException {
    UnderAgeException() {
        super("You are under age");
    }

    UnderAgeException(String message) {
        super(message);
    }
}

public class Voting {
    public static void main(String[] args) {
        int age = 16;
        try {
            if (age < 18) {
                throw new UnderAgeException();
            } else {
                System.out.println("You can vote");
            }
        } catch (UnderAgeException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

vbnet

```
UnderAgeException: You cannot vote as your age is below 18
    at Voting.main(Voting.java:21)
```