

In Java, when using multiple **catch** blocks to handle exceptions, the order of the **catch** blocks matters. The most **specific exceptions** should be **caught first**, followed by more **general exceptions** (like **Exception**, **RuntimeException**, **Throwable**).

**Specific exceptions:** These represent particular errors and are subclasses of **Exception**. (e.g., **NullPointerException**, **ArrayIndexOutOfBoundsException**, **ArithmeticException**)

**General exceptions:** These are broader categories that can catch multiple types of errors. (e.g., **Exception**, **RuntimeException**, **Throwable**)

#### Rule for Ordering Catch Blocks

1. Catch blocks should be arranged from most specific to most general.
2. If a more general exception (like **Exception** or **Throwable**) appears before a more specific exception (like **ArithmeticException**), it will cause a compilation error because the specific catch block becomes unreachable.

#### Example of Correct Order

```
public class MultipleCatchExample {  
    public static void main(String[] args) {  
        try {  
            int a = 10 / 0; // This will throw ArithmeticException  
        } catch (ArithmeticException e) {
```

```

        System.out.println("Arithmetic Exception: " + e.getMessage());
    } catch (Exception e) { // More general exception should be after specific ones
        System.out.println("General Exception: " + e.getMessage());
    }
}
}

```

✓ Output:

Arithmetic Exception: / by zero

Incorrect Order (Compilation Error)

```

try {
    int a = 10 / 0;
} catch (Exception e) { // General exception first
    System.out.println("General Exception: " + e.getMessage());
} catch (ArithmeticException e) { // Specific exception after general one
    System.out.println("Arithmetic Exception: " + e.getMessage());
}

```

✗ Compilation Error:

Exception 'ArithmeticException' has already been caught by the 'Exception' catch block

### Can we catch more than one exception in a single catch block?

Yes, you can catch multiple exceptions in a single **catch** block using **multi-catch** in Java. This was introduced in **Java 7** and allows handling multiple exceptions efficiently, reducing redundant code.

From Java 7, we can catch more than one exception with a single catch block. This type of handling reduces the code duplication. Note : When we catch more than one exception in a single catch block , the catch parameter is implicitly final. We cannot assign any value to the catch parameter. Ex :

```

catch(ArrayIndexOutOfBoundsException || ArithmeticException e) {

}

```

### Example: Handling Multiple Exceptions

```
public class MultiCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {10, 20, 30};
            int result = 10 / 0; // ArithmeticException

            System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException

            int num = Integer.parseInt("abc"); // NumberFormatException

        } catch (ArithmeticException | ArrayIndexOutOfBoundsException | NumberFormatException e) {
            System.out.println("Caught exception: " + e.getClass().getSimpleName());
        }

        System.out.println("Program continues...");
    }
}
```

#### Output:

```
Caught exception: ArithmeticException
Program continues...
```

- The program catches the first exception (`ArithmeticException`) and **does not execute further risky code** inside the `try` block.
- The program **continues execution** after the `catch` block.

### What are checked Exceptions?

1) All the subclasses of the Throwable class except error, Runtime Exception and its subclasses are checked exceptions.

2) Checked exceptions should be thrown with keyword **throws** or should be provided **try catch block**, else the program would not compile. We do get compilation error

. Examples :

- 1) IOException,
- 2) SQLException,
- 3) FileNotFoundException,
- 4) InvocationTargetException,
- 5) CloneNotSupportedException
- 6) ClassNotFoundException
- 7) InstantiationException

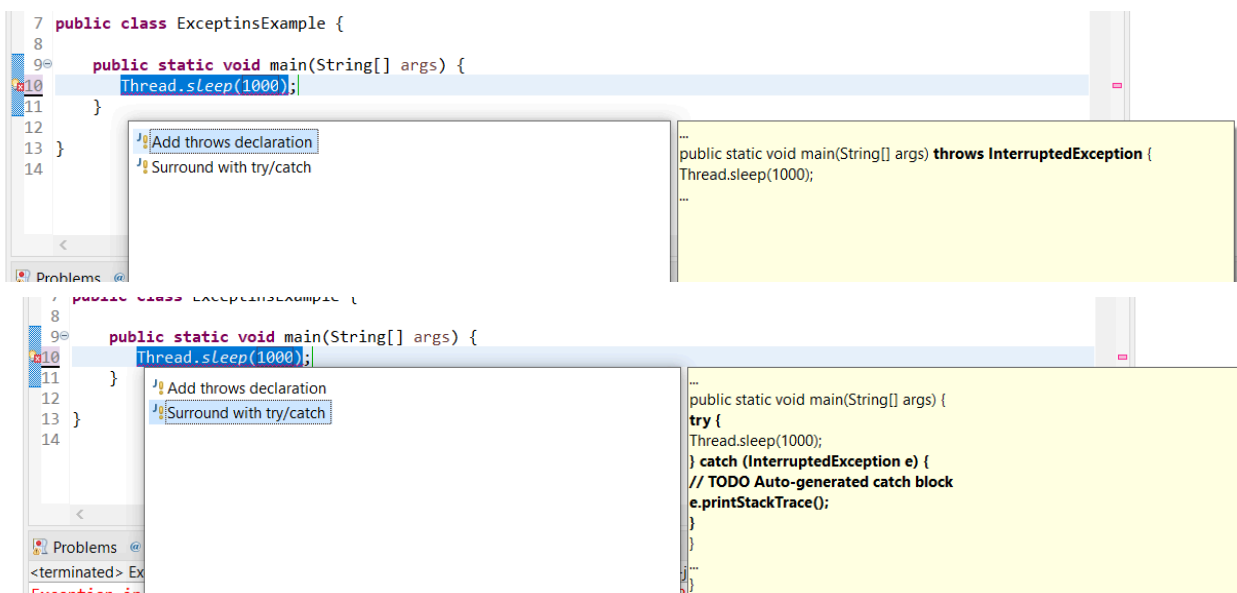
## 8) InterruptedException

Simple checked Exception

**simple checked exception example** using `InterruptedException`:

### Ex1

```
public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            System.out.println("Sleeping for 2 seconds...");
            Thread.sleep(2000); // Throws InterruptedException (Checked Exception)
            System.out.println("Woke up!");
        } catch (InterruptedException e) {
            System.out.println("Caught InterruptedException: " + e.getMessage());
        }
    }
}
```



### Explanation:

- `Thread.sleep(2000);` causes a checked exception (`InterruptedException`).
- The **compiler forces us** to handle this exception using `try-catch` or declare `throws InterruptedException` in the method signature.
- If **not handled**, the code won't compile.

### Ex2:

### SQLException Example

```
import java.sql.*;

public class SQLExceptionExample {
    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "user",
"password");
        } catch (SQLException e) {
            System.out.println("Caught SQLException: " + e.getMessage());
        }
    }
}
```

### What are unchecked exceptions in java?

All subclasses of RuntimeException are called unchecked exceptions. These are unchecked exceptions because the compiler does not check if a method handles or throws exceptions.

Program compiles even if we do not catch the exception or throws the exception. If an exception occurs in the program, the program terminates .

It is difficult to handle these exceptions because there may be many places causing exceptions.

Example :

1. Arithmetic Exception
2. ArrayIndexOutOfBoundsException
3. ClassCastException
4. IndexOutOfBoundsException
5. NullPointerException
6. NumberFormatException
7. StringIndexOutOfBoundsException
8. UnsupportedOperationException

Unchecked Exceptions (Do not need **try-catch** to compile, but cause runtime errors).

**After execution program (compiled successfully) or at Runtime**



```
1 package com.basic;
2
3 import java.sql.*;
4
5 public class ExceptinsExample {
6
7     public static void main(String[] args) {
8         int result = 10 / 0; // Division by zero
9         System.out.println(result);
10    }
11 }
12
13
```

com.b  
Except  
ma

Problems Javadoc Declaration Console  
<terminated> ExceptinsExample [Java Application] C:\Users\jakkula.ramesh\Downloads\eclipse-java-2021-06-R-win32-x86\_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jr  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at com.basic.ExceptinsExample.main(ExceptinsExample.java:8)

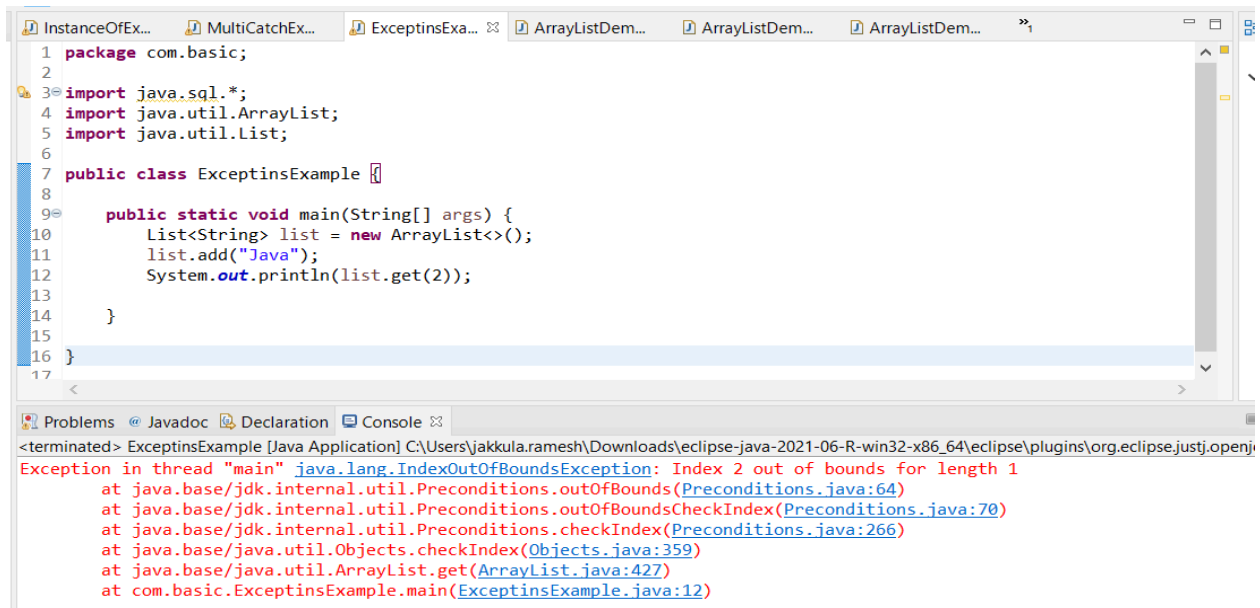
Ex2:

## 2) ArrayIndexOutOfBoundsException Example

```
public class ArrayIndexOutOfBoundsException {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        System.out.println(numbers[5]); // Accessing an invalid index
    }
}
```

**Output:** Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

## 3) IndexOutOfBoundsException Example



```
1 package com.basic;
2
3 import java.sql.*;
4 import java.util.ArrayList;
5 import java.util.List;
6
7 public class ExceptinsExample {
8
9     public static void main(String[] args) {
10         List<String> list = new ArrayList<>();
11         list.add("Java");
12         System.out.println(list.get(2));
13     }
14 }
15
16 }
17
```

Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 2 out of bounds for length 1  
at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)  
at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)  
at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)  
at java.base/java.util.Objects.checkIndex(Objects.java:359)  
at java.base/java.util.ArrayList.get(ArrayList.java:427)  
at com.basic.ExceptinsExample.main(ExceptinsExample.java:12)

#### 4) NullPointerException Example

```
public class NullPointerExceptionExample {  
    public static void main(String[] args) {  
        String str = null;  
        System.out.println(str.length()); // Null reference  
    }  
}
```

**Output:** Exception in thread "main" java.lang.NullPointerException

#### What is default Exception handling in java?

When JVM detects exception-causing code, it constructs a new exception handling object by including the following information. 1) Name of Exception 2) Description about the Exception 3) Location of Exception. After creation of an object by JVM it checks whether there is exception handling code or not. If there is exception handling code then exception handles and continues the program. If there is no exception handling code JVM gives the responsibility of exception handling to the default handler and terminates abruptly. Default Exception handler displays description of exception, prints the stack trace and location of exception and terminates the program. Note : The main disadvantage of this default exception handling is program terminates abruptly.

#### throw Keyword in Java 🚀

The **throw** keyword in Java is used to **explicitly throw an exception** in situations where we need to indicate an error manually. Unlike exceptions that the JVM automatically throws (like **NullPointerException** or **ArithmeticException**), the **throw** keyword allows us to:

```
package com.basic;
```

```
//Custom exception class (Checked Exception)
```

```
class InvalidAgeException extends Exception {  
    public InvalidAgeException(String message) {  
        super(message);  
    }  
}
```

```
public class TryCatchInsideMethod {  
    public static void main(String[] args) {  
        TryCatchInsideMethod validator = new TryCatchInsideMethod(); // Creating object  
  
        validator.validateAge(15); // Calling method with age < 18  
        validator.validateAge(20); // Calling method with valid age  
  
        System.out.println("Program continues...");  
    }  
  
    public void validateAge(int age) {  
        try {  
            if (age < 18) {  
                throw new InvalidAgeException("Age must be 18 or above");  
            }  
            System.out.println("Eligible to vote");  
        } catch (InvalidAgeException e) {  
            System.out.println("Caught Exception: " + e.getMessage());  
        }  
    }  
}
```

### **Explain the importance of throws keyword in java?**

Throws statement is used at the end of method signature to indicate that an exception of a given type may be thrown from the method. The main purpose of throws keyword is to delegate responsibility of exception handling to the caller methods, in the case of checked exception. In the case of unchecked exceptions, it is not required to use throws keyword. We can use throws keyword only for throwable types otherwise compile time error saying incompatible types. An error is unchecked, it is not required to handle by try catch or by throws.

Syntax :



```

Class Test{
Public static void main(String args[]) throws IE {
}
}

```

Note : The method should throw only checked exceptions and subclasses of checked exceptions. It is not recommended to specify exception superclasses in the throws class when the actual exceptions thrown in the method are instances of their subclass

**throws** → Declares that an exception **may occur**, but does **not handle it**.

**try-catch** → **Handles the exception immediately** within the method.

Use **throws** when you want to **delegate (passing the responsibility)** of handling an exception from one method to another.)the exception handling.

Use **try-catch** when you want to **handle** the exception **inside the method**.

### ✓ Key Differences Between throws and try-catch

Feature	throws	try-catch
Purpose	Declares exceptions and passes responsibility to the caller	Handles exceptions inside the method
Used For	Checked exceptions ( <code>IOException</code> , <code>SQLException</code> )	Both checked & unchecked exceptions
Method Responsibility	Just declares exception, does not handle it	Catches and handles the exception within the method
Program Flow	May require handling at multiple levels	Exception is handled immediately
Where Used?	In method signature	Inside method body

### Situation Where **finally** Block Will Not Execute

The **finally** block is **always executed**, except in a few special cases. One such case is when the program is **forcibly terminated** using `System.exit(0)` inside the **try** or **catch** block.

#### ♦ Example: **finally** Block Not Executed

```

class FinallyNotExecuted {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            System.exit(0); // JVM terminates, skipping finally block
        } catch (Exception e) {
            System.out.println("Caught Exception");
        } finally {

```

```
        System.out.println("Inside finally block");
    }
}
}
```

#### ◆ Output

Inside try block

The **finally** block is **not executed** because `System.exit(0)` shuts down the JVM before it gets a chance to run.

#### ✓ Other Situations Where **finally** May Not Execute

1. **JVM Crash or Power Failure**
  - If the JVM crashes (due to an error like `OutOfMemoryError`), the **finally** block **may not** execute.
2. **Infinite Loop or Deadlock in the **try** Block**
  - If an infinite loop or deadlock occurs **before reaching the **finally** block**, it **won't** execute.
3. **Forceful Process Termination**
  - If the process is killed manually (`kill -9` in Unix/Linux or Task Manager in Windows), the **finally** block will **not run**.

## What Are User-Defined Exceptions in Java?

User-defined exceptions are **custom exceptions** created by extending Java's built-in exception classes. They allow us to define **custom error messages** and **specific exception handling logic** for our application.

#### ◆ Types of User-Defined Exceptions

1. **Checked Exceptions** → Extend `Exception` (must be handled using `throws` or `try-catch`).
2. **Unchecked Exceptions** → Extend `RuntimeException` (optional handling, occurs at runtime).

✓ **Best Practice:** It is recommended to create **unchecked** exceptions by extending `RuntimeException` instead of `Exception`, so handling them is optional.

#### ◆ Approach 1: Using Checked Exception (**extends `Exception`**)

This **forces handling** at compile-time using `try-catch` inside the `validateAge()` method.

```
// Custom exception class (Checked Exception)
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CheckedExceptionExample {
    public static void main(String[] args) {
        CheckedExceptionExample validator = new CheckedExceptionExample(); // Creating object
        validator.validateAge(15); // Calling method with age < 18
        validator.validateAge(20); // Calling method with valid age

        System.out.println("Program continues...");
    }

    public void validateAge(int age) {
        try {
            if (age < 18) {
                throw new InvalidAgeException("Age must be 18 or above"); // Explicitly throwing exception
            }
            System.out.println("Eligible to vote");
        } catch (InvalidAgeException e) { // Handling exception
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

#### ♦ **Output (Checked Exception)**

```
Caught Exception: Age must be 18 or above
Eligible to vote
Program continues...
```

 **Exception is handled inside `try-catch`, so the program continues.**

#### ♦ **Approach 2: Using Unchecked Exception (extends `RuntimeException`)**

This **does not require handling** at compile-time, and if unhandled, it stops program execution.

```
// Custom exception class (Unchecked Exception)
class InvalidAgeRuntimeException extends RuntimeException {
    public InvalidAgeRuntimeException(String message) {
```

```

        super(message);
    }
}

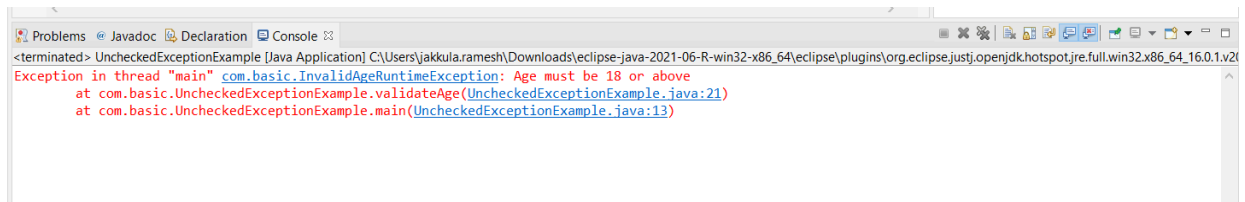
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        UncheckedExceptionExample validator = new UncheckedExceptionExample(); // Creating object
        validator.validateAge(15); // Calling method with age < 18
        validator.validateAge(20); // This won't execute if exception is unhandled

        System.out.println("Program continues...");
    }

    public void validateAge(int age) {
        if (age < 18) {
            throw new InvalidAgeRuntimeException("Age must be 18 or above"); // Throws unchecked
exception
        }
        System.out.println("Eligible to vote");
    }
}

```

#### ◆ Output (Unchecked Exception)



The screenshot shows the Eclipse IDE's console window. It displays a stack trace for an unhandled exception. The message is "Exception in thread 'main' com.basic.InvalidAgeRuntimeException: Age must be 18 or above". The stack trace points to the `validateAge` method in `UncheckedExceptionExample.java` at line 21, and then to the `main` method at line 13. The console window has tabs for Problems, Javadoc, Declaration, and Console.

```

<terminated> UncheckedExceptionExample [Java Application] C:\Users\jakkula.ramesh\Downloads\eclipse-java-2021-06-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.1.v20
Exception in thread "main" com.basic.InvalidAgeRuntimeException: Age must be 18 or above
    at com.basic.UncheckedExceptionExample.validateAge(UncheckedExceptionExample.java:21)
    at com.basic.UncheckedExceptionExample.main(UncheckedExceptionExample.java:13)

```

✗ The program crashes and does not continue because the exception is not handled.

#### Explain the importance of a throwable class and its methods?

Throwable class is the root class for Exceptions.

The **Throwable** class is the **root class** for all exceptions and errors in Java.

- ① **Exception** (Checked & Unchecked Exceptions)
- ② **Error** (Serious problems like `OutOfMemoryError`)

The three methods defined in the throwable class are :

- 1) void printStackTrace() : This prints the exception information in the following format : Name of the exception, description followed by stack trace.
- 2) getMessage() This method prints only the description of Exception.
- 3) toString(): It prints the name and description of Exception

### ◆ 3 Important Methods of Throwable

Method	Description	Example Output
<code>printStackTrace()</code>	Prints <b>full exception details</b> (name, message, stack trace)	<code>java.lang.ArithmeticException: / by zero at Main.main(Main.java:6)</code>
<code>getMessage()</code>	Returns <b>only the exception message</b>	<code>/ by zero</code>
<code>toString()</code>	Returns <b>exception name + message</b>	<code>java.lang.ArithmeticException: / by zero</code>

```

1 package com.basic;
2
3 public class ThrowableExample {
4     public static void main(String[] args) {
5         try {
6             int result = 10 / 0; // ArithmeticException
7         } catch (ArithmeticException e) {
8             System.out.println(" Using getMessage(): " + e.getMessage()); // Prints only the message
9             System.out.println(" Using toString(): " + e.toString()); // Prints name + message
10            System.out.println(" Using printStackTrace():");
11            e.printStackTrace(); // Prints full stack trace
12        }
13    }
14 }
15

```

```

<terminated> ThrowableExample [Java Application] C:\Users\jakkula.ramesh\Downloads\eclipse-java-2021-06-R-win32-x86_64\eclipse\plugins\org.eclipse.justjop
Using getMessage(): / by zero
Using toString(): java.lang.ArithmeticException: / by zero
Using printStackTrace():
java.lang.ArithmeticException: / by zero
    at com.basic.ThrowableExample.main(ThrowableExample.java:6)

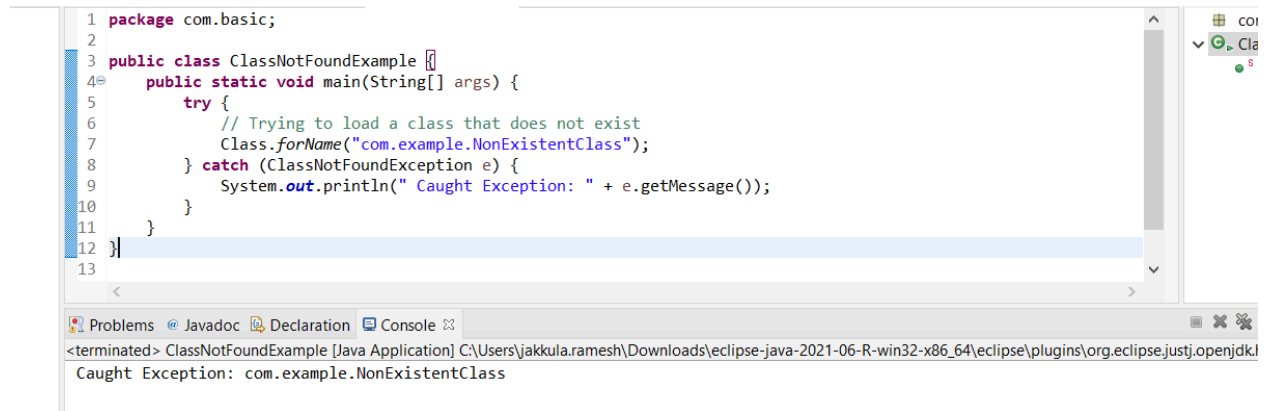
```

## When Will `ClassNotFoundException` Be Raised?

`ClassNotFoundException` occurs when **JVM** tries to load a class by its string name but cannot find it in the classpath.

- ## ◆ Common Scenarios That Cause `ClassNotFoundException`

- ❶ Misspelled Class Name
- ❷ Class File Not Available in Classpath
- ❸ Trying to Load a Class Dynamically Using `Class.forName()`
- ❹ JAR File Missing from Classpath



The screenshot shows the Eclipse IDE with a Java file named `ClassNotFoundExample.java` in the package `com.basic`. The code attempts to load a class that does not exist using `Class.forName("com.example.NonExistentClass")`. The console output shows the exception message: `Caught Exception: com.example.NonExistentClass`.

```
1 package com.basic;
2
3 public class ClassNotFoundExample {
4     public static void main(String[] args) {
5         try {
6             // Trying to load a class that does not exist
7             Class.forName("com.example.NonExistentClass");
8         } catch (ClassNotFoundException e) {
9             System.out.println(" Caught Exception: " + e.getMessage());
10        }
11    }
12 }
13
```

Console Output: `<terminated> ClassNotFoundExample [Java Application] C:\Users\jakkula.ramesh\Downloads\eclipse-java-2021-06-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk`  
`Caught Exception: com.example.NonExistentClass`

Ex2:

`Class.forName("com.basic.ClassNotFoundExample")` will NOT throw `ClassNotFoundException` because the class is already available in the classpath.



The screenshot shows the Eclipse IDE with a Java file named `ClassNotFoundExample.java` in the package `com.basic`. The code attempts to load the class `com.basic.ClassNotFoundExample` using `Class.forName("com.basic.ClassNotFoundExample")`. The console output shows the exception message: `Caught Exception:` .

```
1 package com.basic;
2
3 public class ClassNotFoundExample {
4     public static void main(String[] args) {
5         try {
6             // Trying to load a class that does not exist
7             Class.forName("com.basic.ClassNotFoundExample");
8         } catch (ClassNotFoundException e) {
9             System.out.println(" Caught Exception: " + e.getMessage());
10        }
11    }
12 }
13
```

Console Output: `<terminated> ClassNotFoundExample [Java Application] C:\Users\jakkula.ramesh\Downloads\eclipse-java-2021-06-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdkhotspot.jre.full.win32.x86_64_16.0.1.v202105`  
`Caught Exception:`