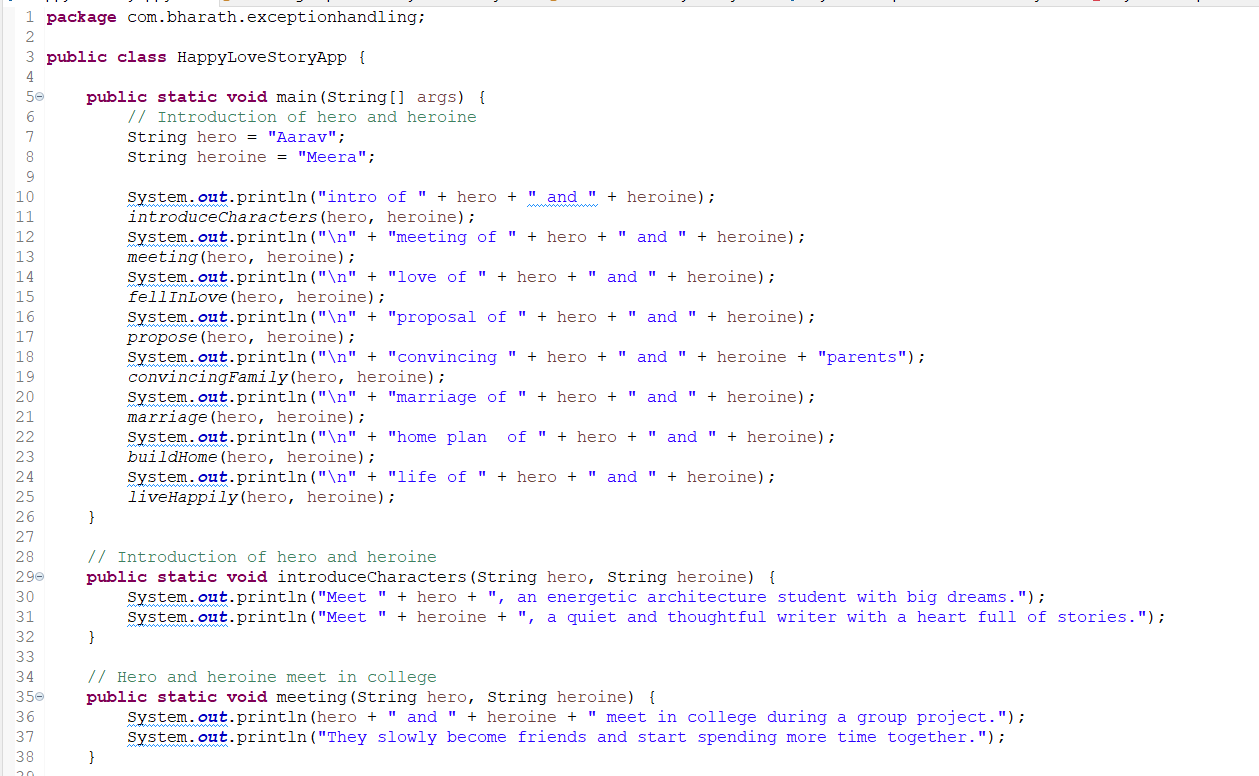
Story  
**Happy Story:**

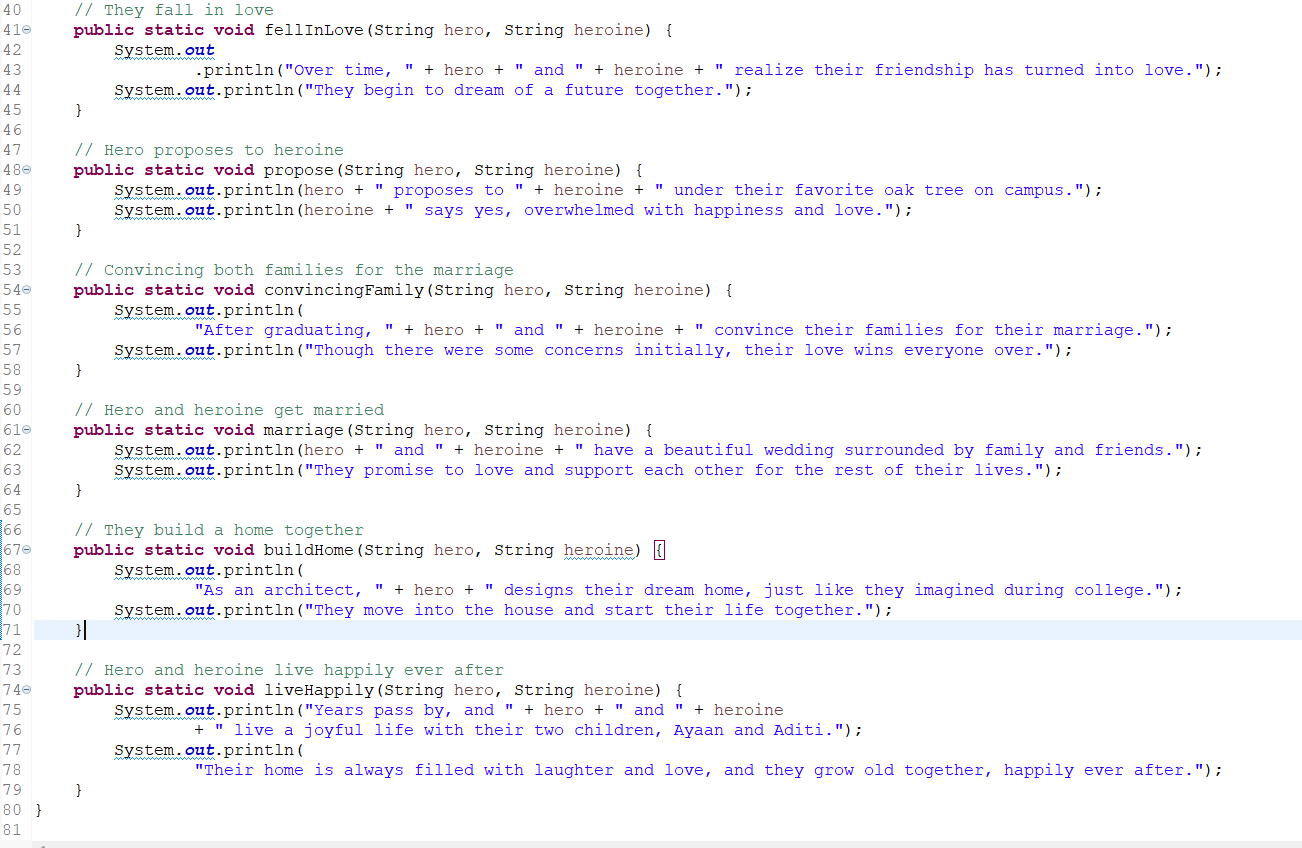
Aarav and Meera met in college and fell in love as they dreamed of their future. Aarav, an energetic architecture student, would often sketch their future home, imagining a life filled with love, children, and laughter. Meera, a quiet writer, loved how he could see their future so clearly. After graduating, they got married, built their dream home, and welcomed two children, Ayaan and Aditi. Their days were filled with joy, and as they grew old together, they cherished the beautiful life they had created, just as they had once planned under the oak tree.

**Tragedy Story:**

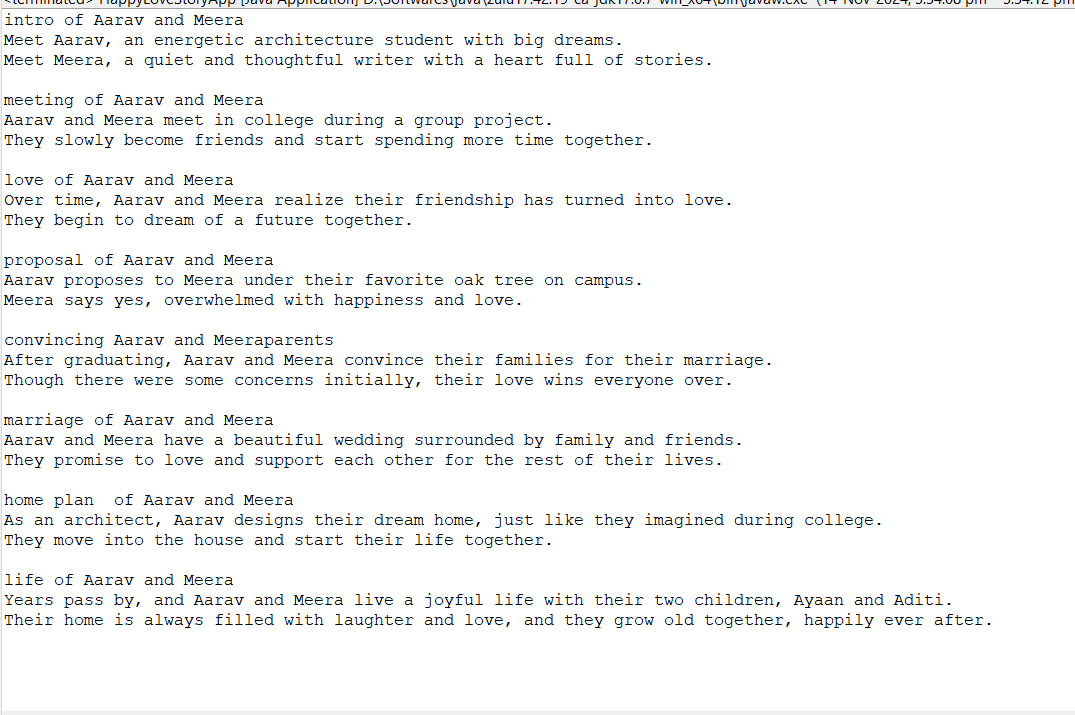
Aarav and Meera met in college and dreamed of a future together. Aarav, an adventurous architecture student, loved designing their future home, while Meera, a thoughtful writer, imagined the life they would build with their children. One day, after graduating, they decided to convince their families for their marriage. Aarav, eager and excited, was on his way to meet Meera’s parents when he rushed out without wearing the helmet Meera always insisted he wear. In his haste, he forgot. That day, Aarav met with a tragic accident. Meera’s world shattered as all the dreams they had—building a home, raising children—vanished. Left with only memories, Meera grieved the future they would never have, holding on to the love they shared.

Lovestory app

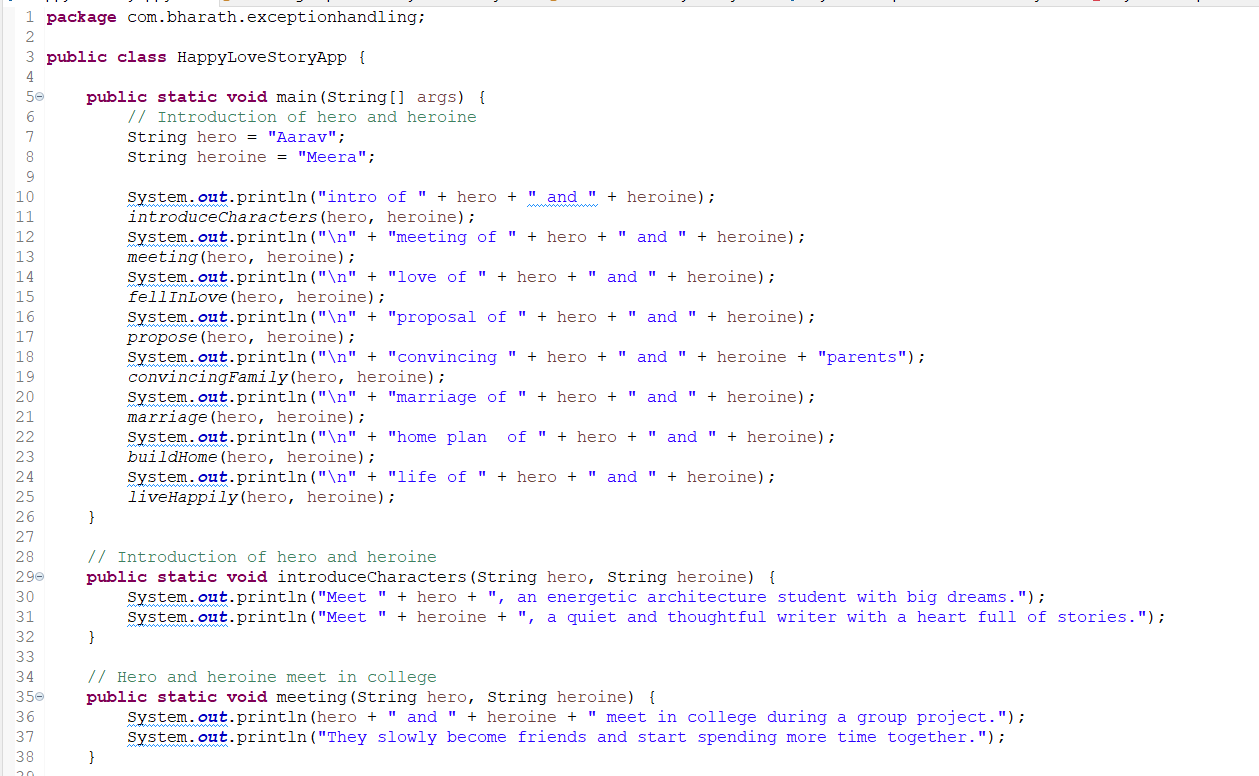


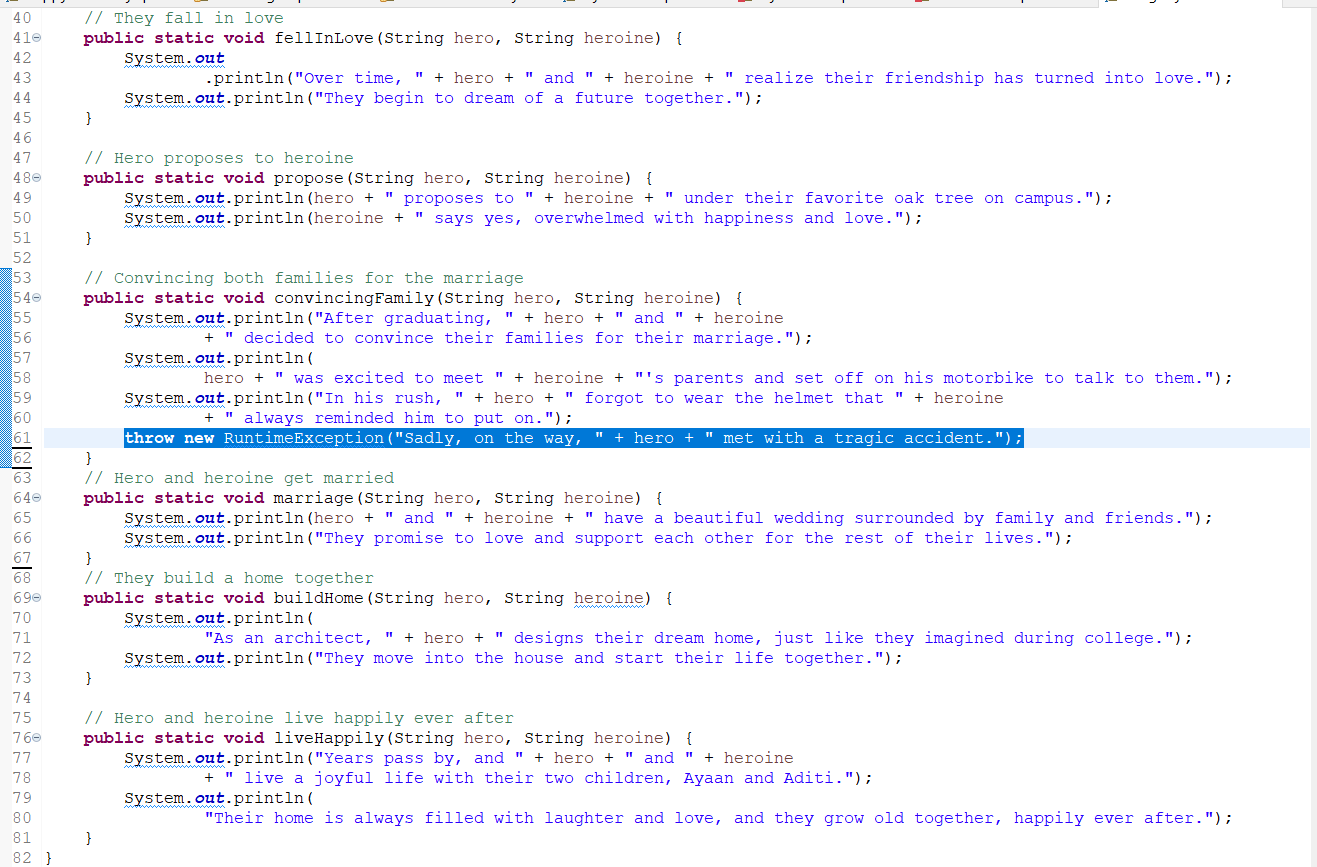


Output:

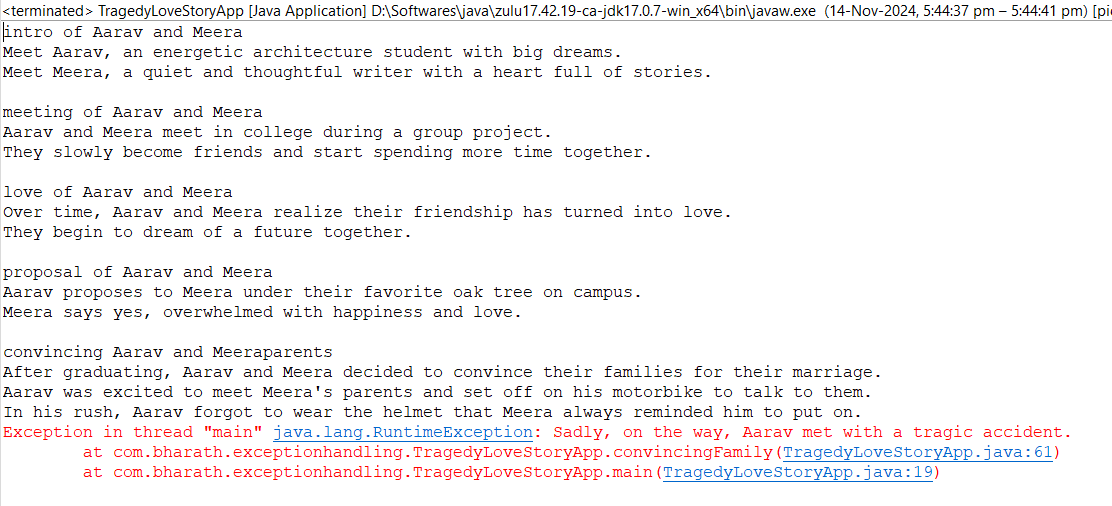


Tragedy





Output



Normal application flow

**JVM Initialization:**

* When a Java program is executed, the **JVM** is loaded, and it sets up the environment needed to run the program.
* The JVM starts by creating an initial **main thread** which will be responsible for executing the application.

**Creating a Stack for the Main Thread:**

* The JVM creates a **call stack** for the **main thread**. The call stack is a data structure that keeps track of method calls and local variables for each thread.
* The **main thread** is the first thread to start and will execute the main() method of the class containing it.

**Calling the main() Method:**

* The **main method** (public static void main(String[] args)) is the entry point of a Java application. The JVM calls this method when the program starts.
* This is the point where the execution begins.

**Method Calls and Stack Frames:**

* As the program executes and calls methods, the JVM adds each method call to the **stack** in the form of **stack frames**.
* A stack frame is created for each method call, containing the local variables and the method’s execution context (the current state of the method).

**Method Execution:**

* The methods that are called from main() (or any other method) will be executed one by one. During the execution, each method gets its own stack frame to hold local variables and the method's state.

**Completion of Method Execution:**

* Once a method completes its execution, its stack frame is removed from the stack, and the control is returned to the method that called it.
* If the method was the last one called in a chain, the program proceeds to the next statement or ends the program if there are no further instructions.

**Thread Completion:**

* Once all methods have executed and there are no more instructions to execute, the **main thread** completes its execution.
* The **main thread** is now **terminated**, and the **JVM** begins to clean up.

**Program Termination:**

* After the main thread finishes, if no other threads are running, the JVM terminates the program.
* The JVM releases resources, and the program ends.

What is exception?

Imagine you're following a beautiful story, but suddenly, something unexpected happens that disrupts the flow, causing the characters' plans to fall apart. This sudden disruption in the story represents what we call an **exception** in Java.

In Java, an **exception** is an unexpected event or error that occurs during the execution of a program, which can disrupt its normal flow. When an exception occurs, the program can’t continue as planned, and Java throws an exception to indicate that something went wrong.

What happens if an exception occurs? Abnormal termnation

In normal program termination, after the execution of each method, the context in the stack frame will be removed from the stack. However, if an exception occurs while executing any method, that method is responsible for creating the respective **exception object** and it sets the stacktrace info using(fillInstackTrace() method). Once the exception object is created, the JVM will check whether the method is handling the exception (using a try-catch block).

If the respective method does **not** handle the exception, the rest of the code in that method will not execute, and its stack frame will be removed from the stack. The JVM will then check with the **caller method** to see if it handles the exception. If the caller method is also not handling the exception, its stack frame will also be removed from the stack.

This process continues up the call stack until the **main method**. If the main method also does not handle the exception, then the responsibility to handle the exception is taken over by the JVM. Since the JVM is responsible for calling the main method, it will use its **default exception handler** to handle the exception. The JVM then prints the complete exception stack trace, showing where the exception occurred and the path it followed up the call chain.

Default exception handler by JVM

JVM will **print the exception stack trace** and **terminate the program abnormally**. This means the program doesn't follow the usual exit flow and is instead halted due to the error.

**Exception Stack Trace:**

The **stack trace** provides detailed information about the exception, which helps in debugging. It includes:

1. **Type of Exception**: The class name of the exception (e.g., NullPointerException, ArrayIndexOutOfBoundsException).
2. **Description of the Exception**: A brief message about what went wrong, often provided by the exception message (e.g., "Index 5 out of bounds for length 4").
3. **Location of the Exception**: The list of method calls (stack frames) that led to the exception. This includes:
   * The method where the exception occurred.
   * The class and line number in the source code where the exception was thrown.
   * A chain of method calls showing how the program arrived at the point of failure.

Exception Hireachy

Throwable is the parent class for all exceptions in java.lang package where no exception is not be be imported

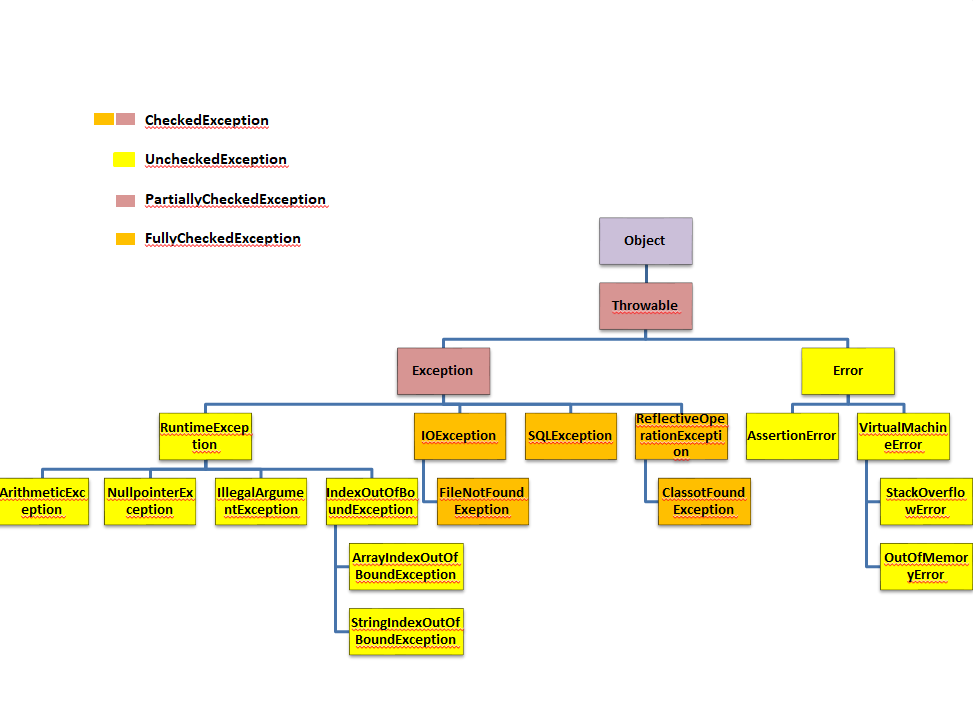


Figure:Exception Hirearchy

NOTE**: All exceptions occur at runtime**

Checked Exception   
  
Lets have a scnerio to understand the checked exception

### Scenario 1:

Imagine you’re planning a trip, and on the way, your car’s tire unexpectedly goes flat. Unfortunately, you don’t have a spare tire, and there are no repair shops nearby, which means your trip is ruined.

**What do we do next time when we plan a trip?**  
We’ll check for the spare tire, right?

The next time you plan a trip, you decide to be more cautious and check if you have a spare tire. But this time, your gas runs out, and there’s no gas station nearby. The trip is interrupted again.

**What do we do next time when we plan a trip?**  
For the next trip, you make sure to have both a spare tire and enough gas, right?

### Scenario 2:

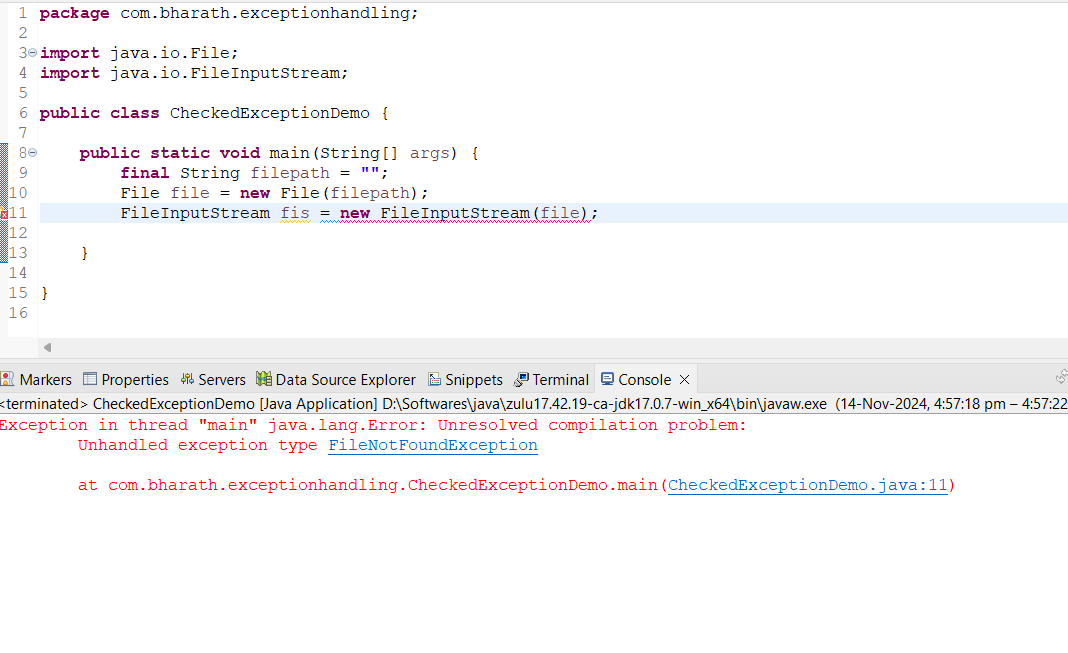
Imagine you have an exam in a difficult subject, so you’re in a rush revising the topics and leave for the exam. Once you reach the exam hall, the examiner asks for the hall ticket. Unfortunately, you don’t have it, and the examiner doesn’t allow you to write the exam.

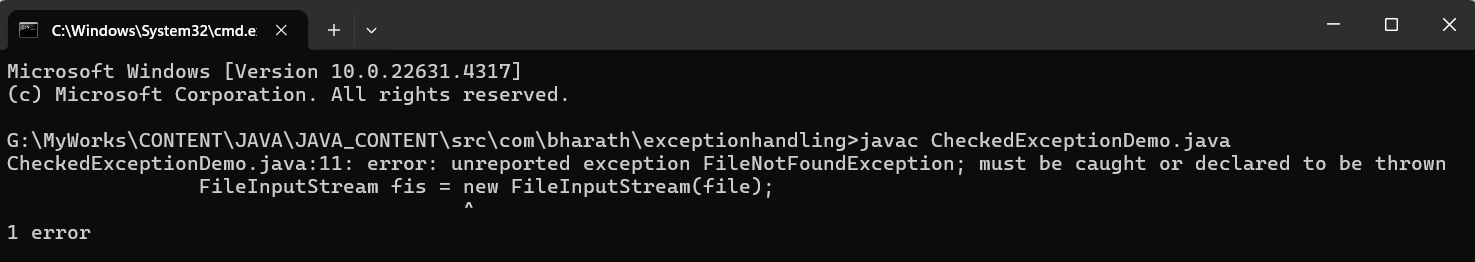
**So next time when you have an exam, what will you do?**  
Before going, you will check for the hall ticket, right? You’ll also remind others to carry their hall tickets before leaving for the exam hall.

Like the scenarios above, where we become cautious to avoid the same interruptions in the future. In the same way, there are certain exceptions in Java where their occurrence is possible, so these frequently occurring exceptions can be detected by the compiler which are called as **CheckedExceptions**(refer Fig Exception Hirearchy).

For example, when dealing with file operations (opening and closing), there might be a chance of a **FileNotFoundException**, which the compiler forces us to handle. Similarly, when we are connecting to a database, an **SQLException** might occur, so the compiler asks us to handle it. When we load a class using Class.forName, there is a chance of a **ClassNotFoundException**, which can also be detected by the compiler and needs to be handled.

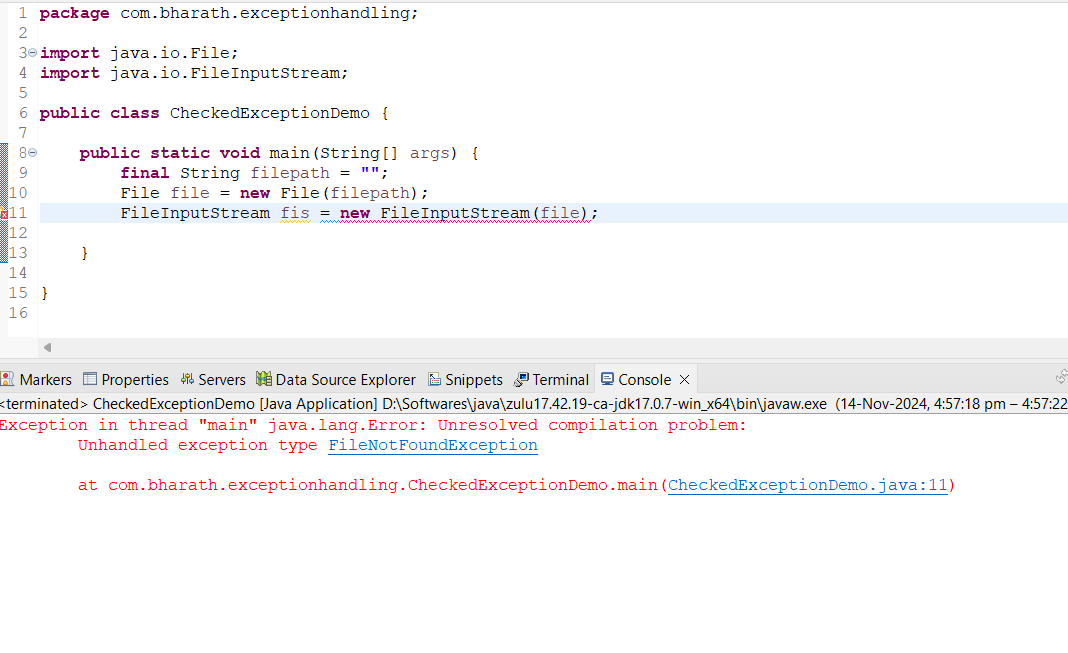
Demo for reading a file





Explaination

Compiler detects the chance of occurance of FileNotFoundException which is to be handled else it considers as a compilation error

Output

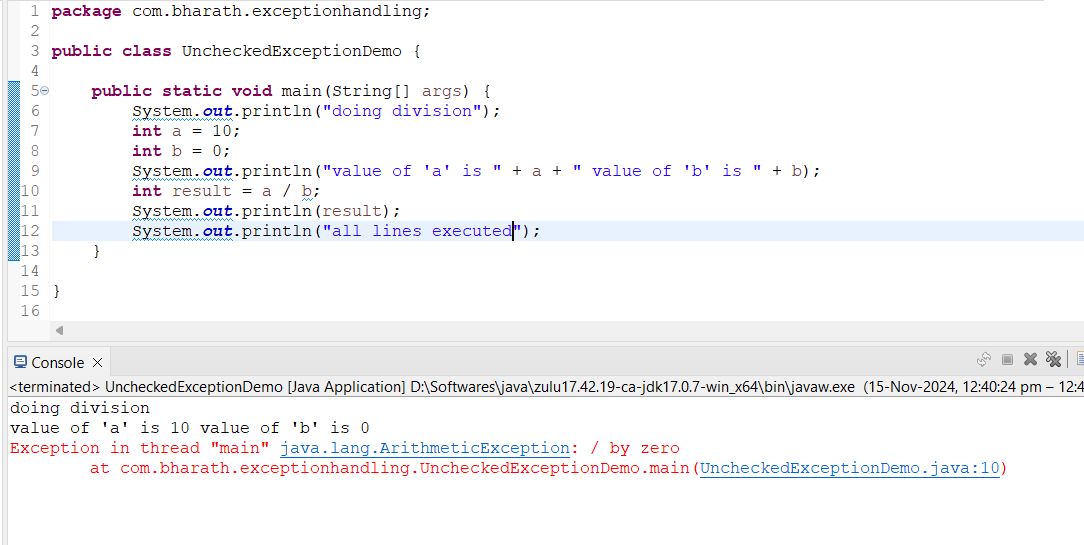
Unchecked exception

The exception that cannot be detected by the compiler and can only be found at runtime, during the execution of the program, is called an **unchecked exception** (refer to the Exception Hierarchy).

Unchecked exceptions are typically caused by programming errors, such as logical mistakes or improper use of APIs.

For example issues like dividing by zero causes **ArithmeticException**, accessing invalid array indices causes **ArrayIndexOutOfBoundException**, or performing operations on null object causes **NullpointerException**As unchecked exceptions are cannot be detected by the compiler the program might get terminated abnormally like discussed earlier

ArithmeticException demo



In the above example, **line 12** is not executed because we encountered an **ArithmeticException** while executing **line 10**. Since we cannot divide by zero, the JVM automatically creates an **ArithmeticException** object internally. It sets the stack trace using the fillInStackTrace() method and throw the exception, and this same stack trace is printed to the console.

The program then stops executing further, resulting in **abnormal termination**. This means that the rest of the code does not execute after the exception is thrown, and the program terminates without completing successfully.

**Why Handle Exceptions?**

We handle exceptions to:

1. **Avoid compilation errors** in the case of **checked exceptions**.
2. **Prevent abnormal termination** in the case of **unchecked exceptions**.
3. To give a proper message to the end user

Proper exception handling ensures that the program can recover from errors and continue execution without crashing.

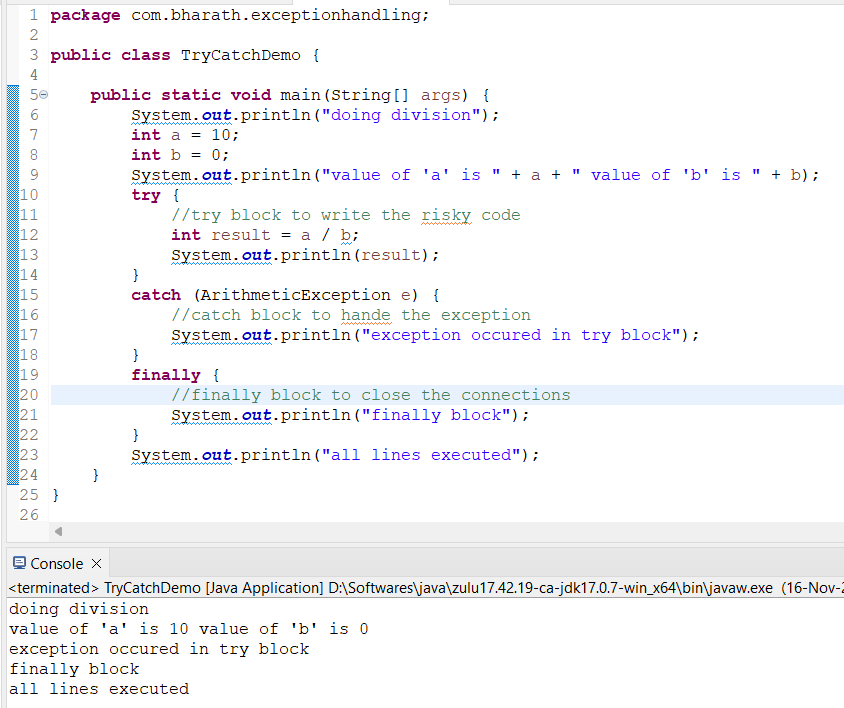
**How to Handle an Exception?**

In Java, exceptions are handled using the **try-catch** blocks.

* **Try block**: Contains the **risky code**, i.e., the code that might throw an exception. For example loading a class,connecting to db,reading a file etc
* **Catch block**: Handles the exception if one occurs, preventing the program from terminating abnormally.

NOTE: try and catch are keywords of java

Lets understand the use of try catch blocks by modifying the unchecked exception demo and checked exception demo



In the above example, we placed the arithmetic operation in the try block. In **line 12**, we are performing division by zero, which creates an **ArithmeticException** object. This exception is thrown, and because we used a catch block, instead of abnormal termination, the JVM transfers control to the catch block. It then executes the code inside the catch block (in **line 16**). After the catch block finishes, the JVM continues with the next lines of code after the catch block, i.e., it executes **line 18**.

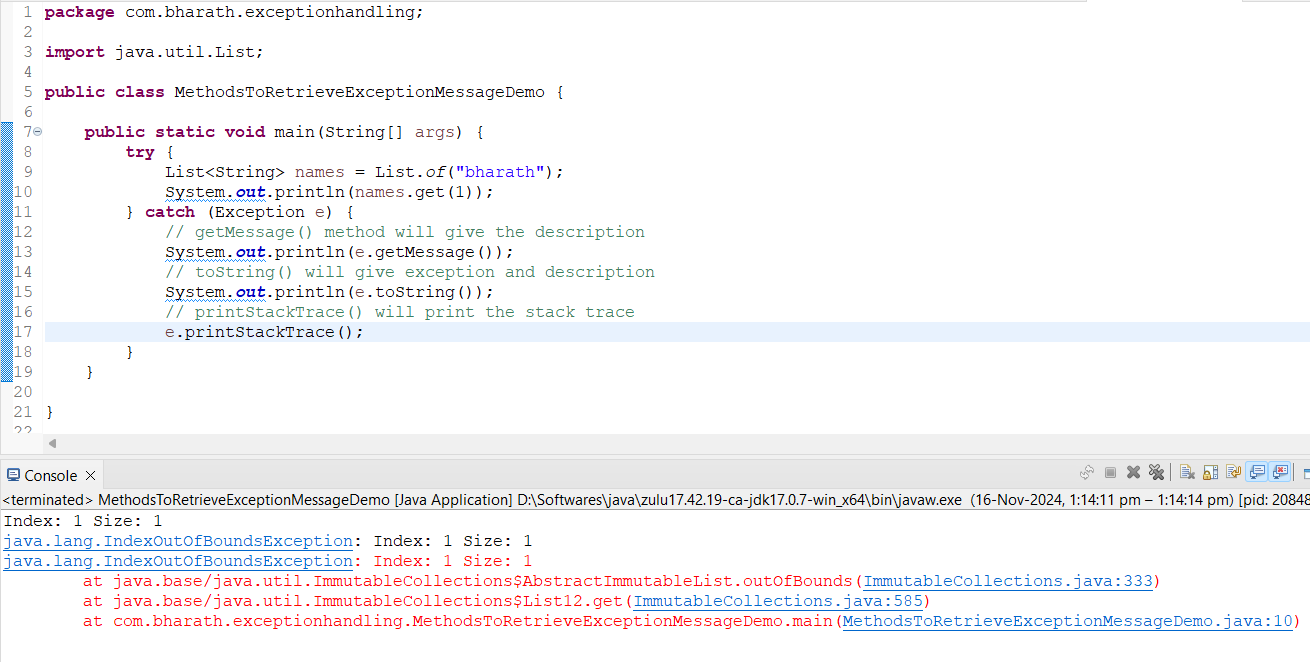
**Finally block**: It is used to handle tasks related to closing resources such as file streams, database connections, or releasing other system resources. The finally block ensures that certain important code is executed, regardless of whether an exception occurs or not.

**Note:** Even though we used a catch block, it doesn't mean that all lines in the try block get executed. For instance, in this example, **line 13** will not get executed because the exception occurred in **line 12**, and the program immediately jumps to the catch block without continuing in the try block.

Once we get the exception we can get the reason/description by using the Throwable class methods

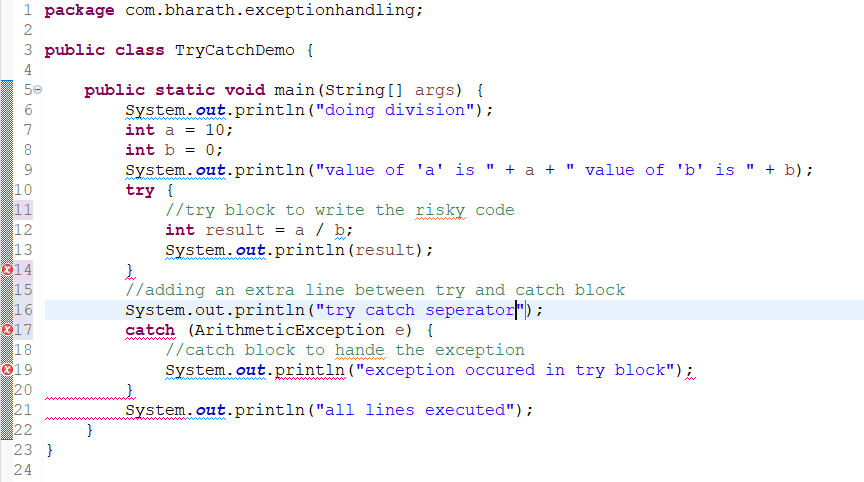
**Methods to Retrieve Exception Details:**

1. **printStackTrace()**:
   * This method prints the complete stack trace of the exception to the standard error output (System.err).
   * It provides the full history of the method calls leading to the exception, starting from the point where the exception was thrown and moving up through the call stack.
2. **toString()**:
   * This method returns a string representation of the exception.
   * It contains the **name of the exception class** and the **description/message** of the exception.
3. **getMessage()**:
   * This method returns the **detailed message** of the exception (the description of what went wrong).
   * It is useful when you only need the error message and not the full stack t



NOTE: JVM uses printStackTrace() method to handle the exception

Key points :  
try block should be immediately followed by the catch block/finally block or both else we get the compilation error  
we cannot have catch block independently / without try block



In above example we got two compilation errors because of line 16  
as we added line 16 try is not immediately followed by catch or finally block  
catch block is independent in line 17

Fully and partially checked exception





Customised exception  
  
**Scenario:**

Arun is a developer working on an e-commerce application. He wrote an API to fetch product details, but unexpectedly, the product is not available, causing a **NullPointerException**. After investigating, Arun realizes that instead of letting a NullPointerException reach the user, it would be better to show a meaningful message, like "Product not found." How can he achieve this?

**Solution:**

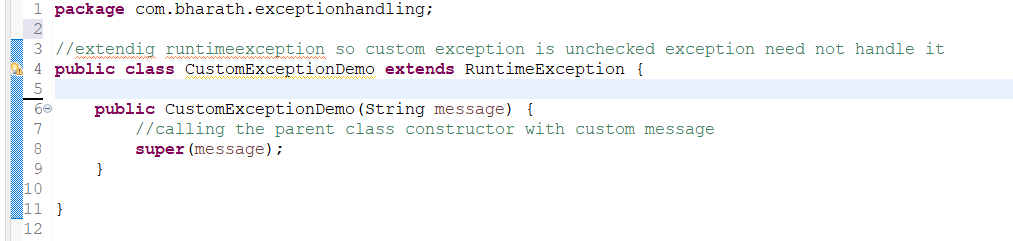
Yes, there is a way to handle this in Java by using **custom exceptions**. This approach allows you to define meaningful, domain-specific exceptions, such as:

* ProductNotFoundException
* AmountNotAvailableException
* UserNotFoundException

**How to Create a Custom Exception in Java:**

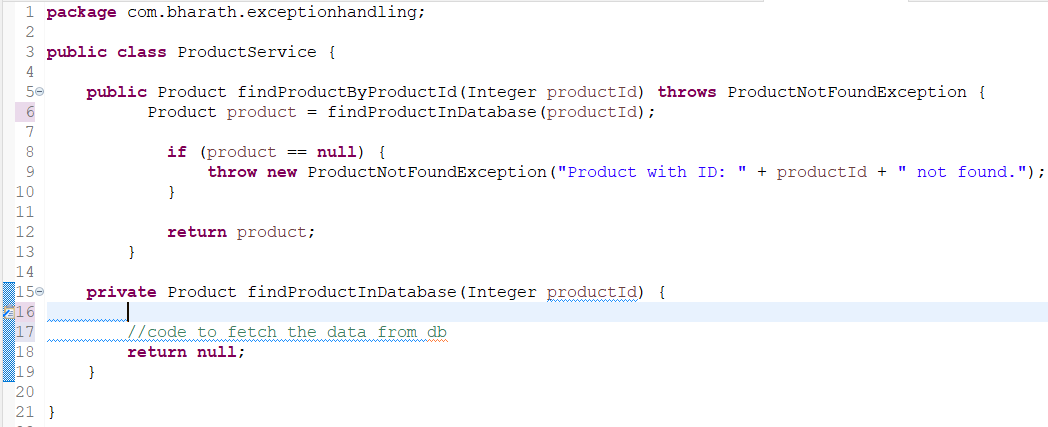
You can create a custom exception by extending the **Throwable** class or its subclasses, especially **RuntimeException** (for unchecked exceptions) and **Exception** (for checked exceptions).

**Example:**

**How Custom Exceptions Work:**

1. **Extending Throwable**: By extending Throwable or one of its subclasses (Exception or RuntimeException), the custom exception inherits the standard properties and behaviors of exceptions.
2. **Throwing the Custom Exception**: When an error scenario (like a missing product) is encountered, you can throw your custom exception instead of allowing a generic exception (like NullPointerException) to propagate.
3. **Catching the Custom Exception**: The JVM will catch the custom exception as it does with any standard exception. You can then handle it appropriately, either showing a meaningful message to the user or performing some other action.

**Example Usage in Code:**

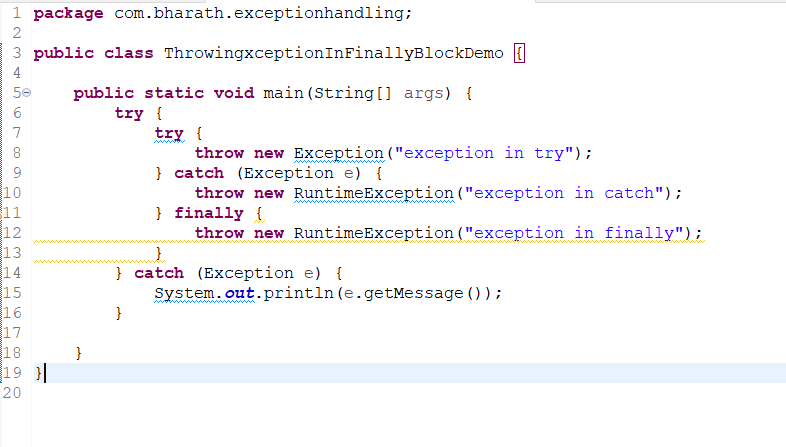
In this example:

* When the product is not found in the database, the custom ProductNotFoundException is thrown with a meaningful message.
* This custom exception can then be caught and handled by the calling method, ensuring that the user receives a friendly message instead of a raw stack trace.

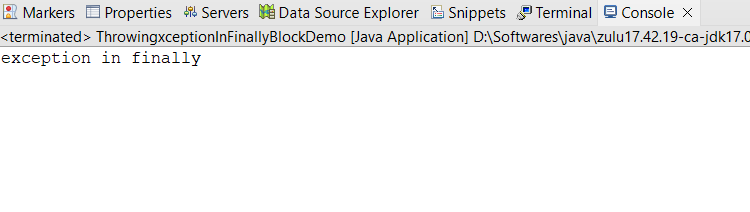
Throw and throws key word

Exception cases

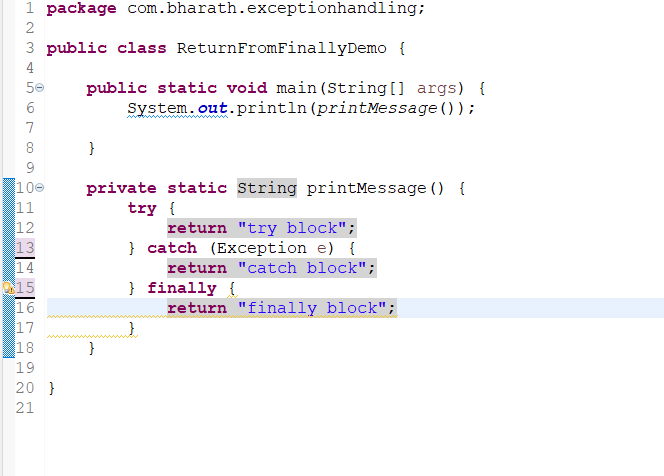
case



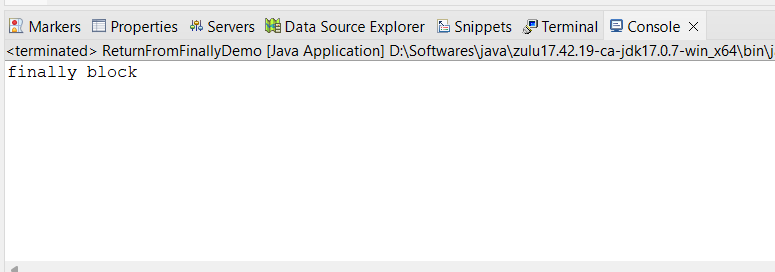
Output:



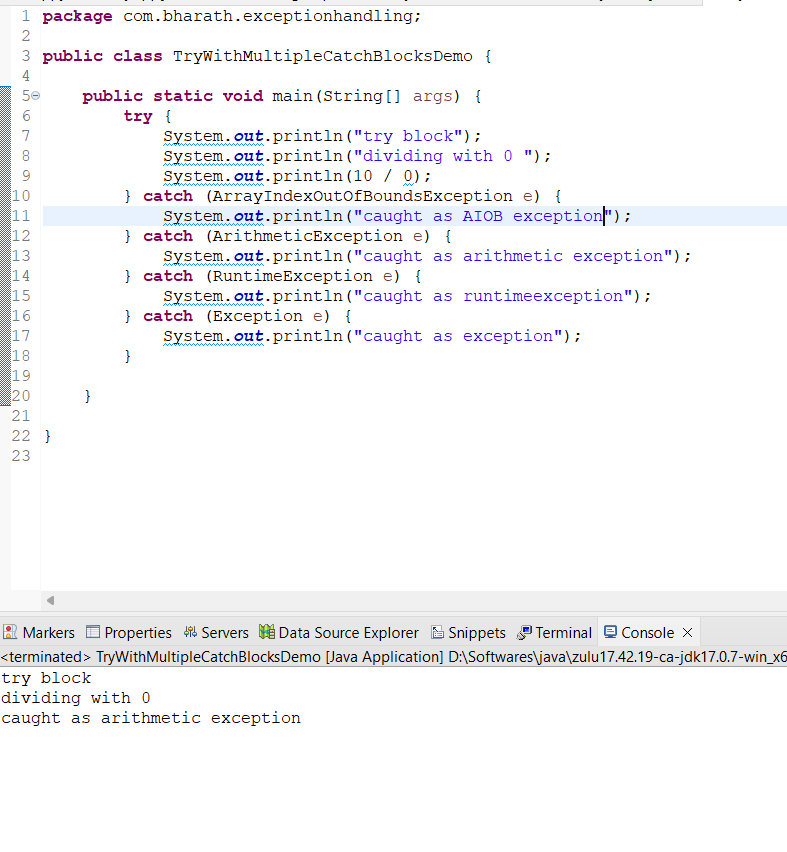
Case



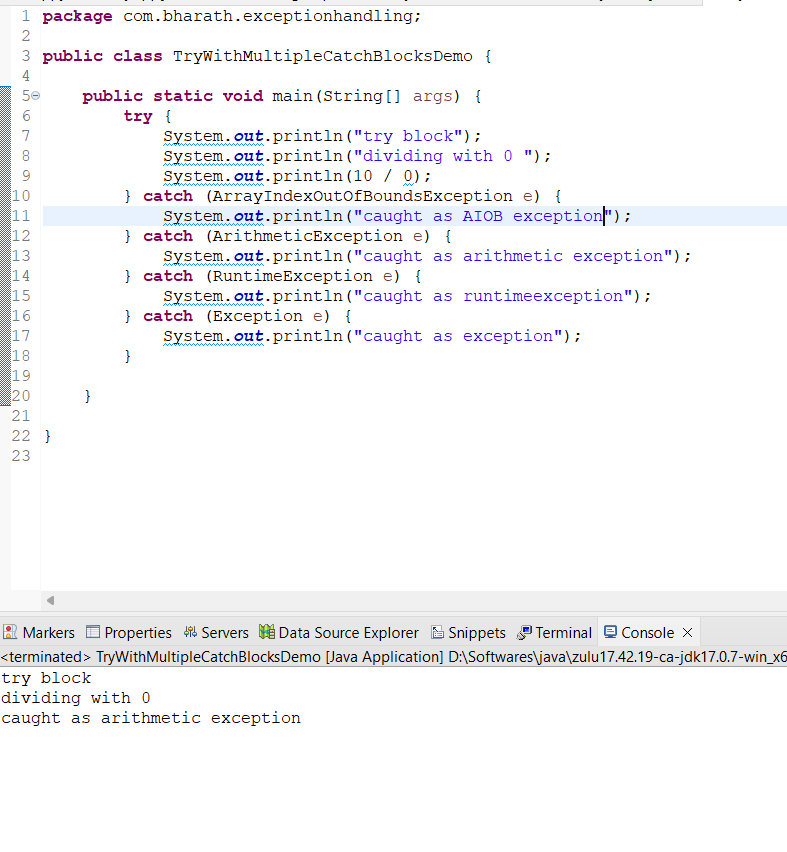
Output



Case



Output



Case



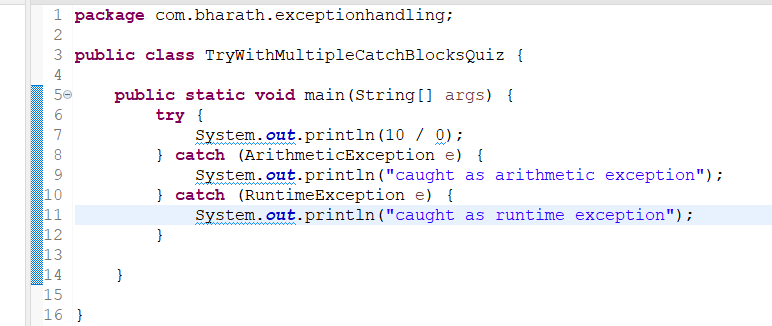
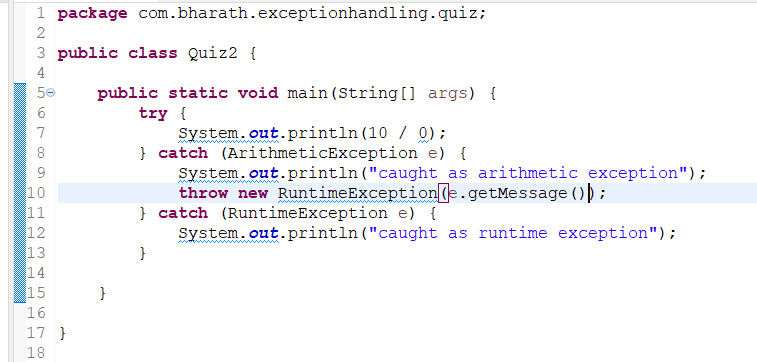
Output



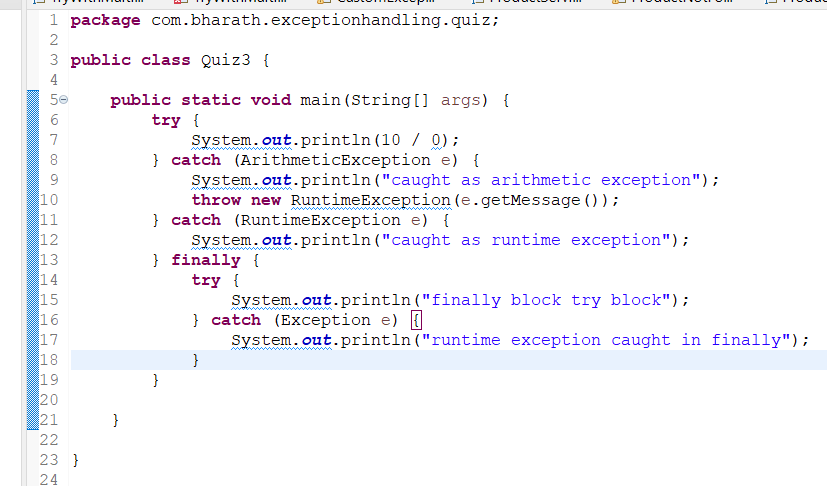
Explaination

According to hierarchy Exception class is parent to the RuntimeException class   
So according to try with multiple catch blocks parent exception class should be thelast catch block if it is in between then the child class exception would not get chance to handle the exception

Quiz

ans:  


3



Ans

interview questions

1.what is the difference between throw and throws keyword?

A.Throw: The throw keyword is used within a method to explicitly create and throw an exception. It notifies the JVM to handle the exception by creating an exception object.

Throws: The throws keyword is used in the method declaration to indicate that the method may throw an exception and that the calling method is responsible for handling it. It is used to propagate the exception to the caller method.

2.How try-with resources work?

A.The **try-with-resources** statement ensures that resources (like file streams, database connections, etc.) are automatically closed after the program is done using them, whether an exception occurs or not. You don’t need to explicitly close the resource in a finally block. The resources must implement the **AutoCloseable** or **Closeable** interface for this to work.

3.What is Exception?

A.

4.What is difference between checked exception and unchecked exception?

5.What is finally block?

6.Can we have multiple catch bocks and its use?

7.can try block exists without a catch block?