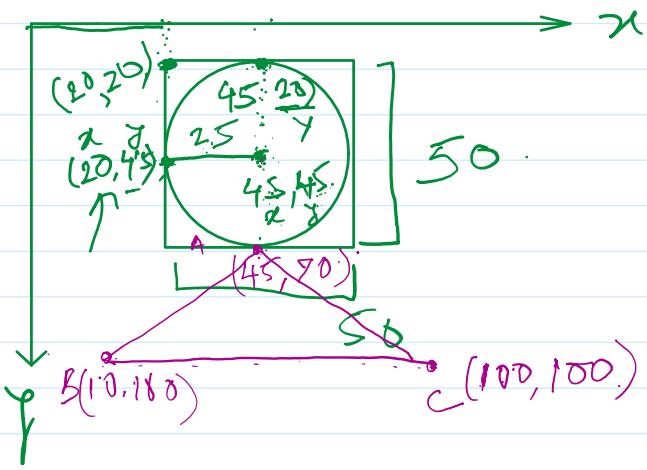



```
BlueJ: Terminal Window - TTcs
Room length: 10.5
Expected: 10.5
Room width: 9.0
Expected: 9.0
Surface area: 360.08
Expected: 368.72
Paint cost: 38.35
Expected: 39.27
Total cost: 138.35
Expected: 139.27
Room length: 15.0
Expected: 15.0
Room width: 12.0
Expected: 12.0
Surface area: 702.08
Expected: 574.22
Paint cost: 74.77
Expected: 61.15
Total cost: 174.77
Expected: 161.15
```

10.5 * 9 * 4 = 378
378 - ((80*32)/144) + 20) =
378 - ((80*32)/144) + 20) =

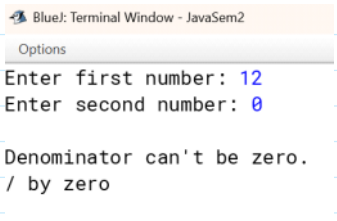
Exception Handling In Java

30 April 2025 20:22

The **exception handling** in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class GeneratingException here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class GeneratingException
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      int a, b, r;
      try{
         System.out.print("Enter first number: ");
         a = sc.nextInt();
         System.out.print("Enter second number: ");
         b = sc.nextInt();

         r = a / b;

         System.out.print("\nQuotient = " + r);
         r = a % b;
         System.out.print("\nRemainder = " + r);
      }
      catch(ArithmeticException ae){
         System.out.print("\nDenominator can't be zero.\n" + ae.getMessage());
      }
   }
}
```

Output:



What is Exception in Java?

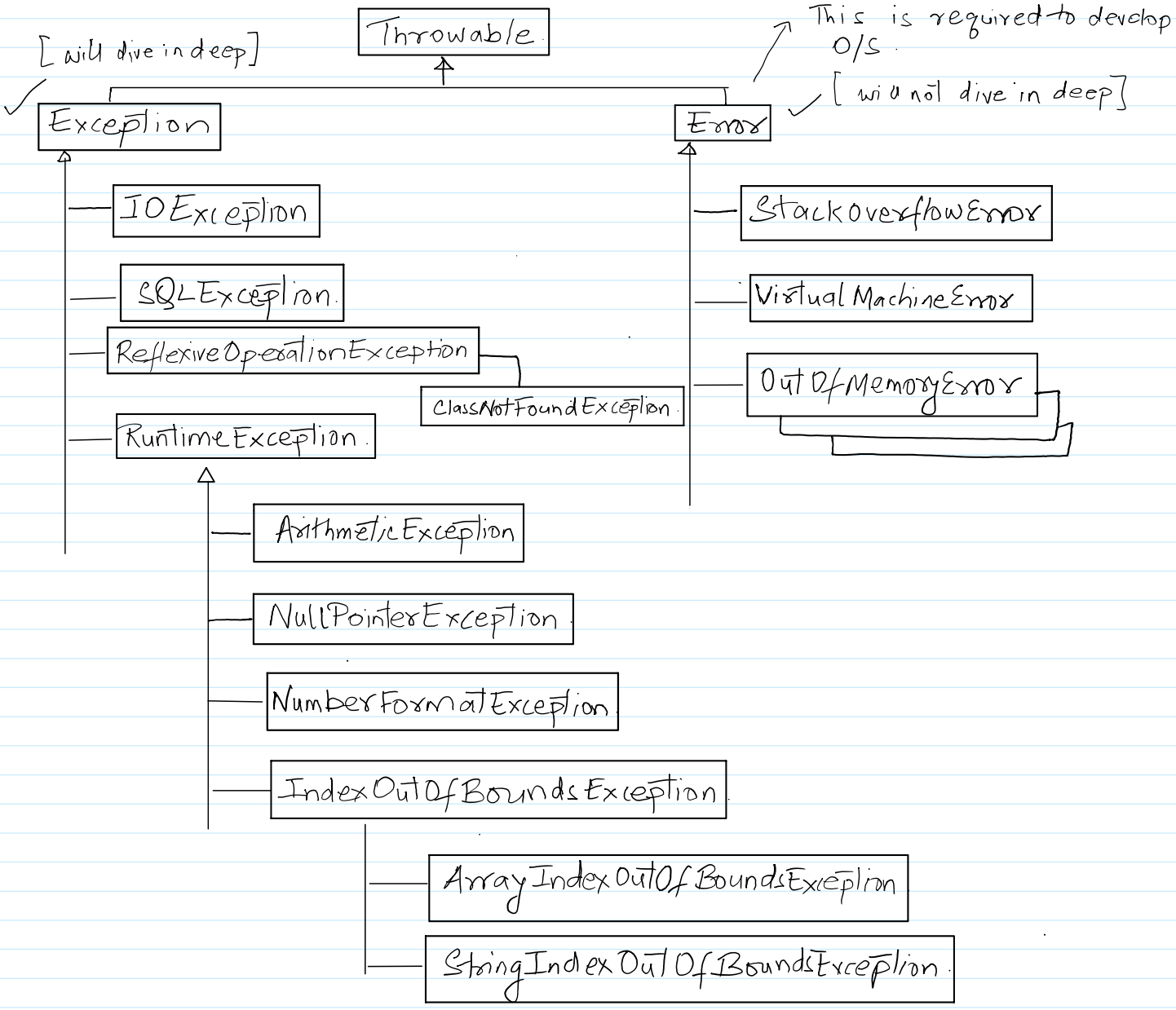
Tin Java, an exception is an event that occurs during the execution of a program that. disrupts the normal flow of instruction. These exceptions can occur for various reasons, such as invalid user input file not found or division by zero when an exception occurs. It is typically represented by an object of a subclass of the **java.lang.Exception** class.

What is Exception Handling?

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application That is why we need to handle exceptions.

Hierarchy of Java Exception Classes

The **java.lang.Throwable** class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java exception class is given below:



Types of Exceptions

In Java exceptions are categorized into two main types. Checked Exceptions and Unchecked exceptions. Additionally, there is a third category known as errors.

What is Checked Exceptions?

Java exceptions are the exceptions that are checked at compile time. This means that the compiler verifies that the code handles these exceptions either by catching them or declaring them in the method signature using the throws keyword. Example of checked exceptions include:

IOException
SQLException
ParseException
ClassNotFoundException, etc

What is Unchecked Exceptions?

Uncheck the exceptions, also known as the runtime exceptions are not. checked at compile time These exceptions usually occur due to programming errors such as logic errors or incorrect assumptions in the code. They do not need to be declared in the method signature using the **throws** keyword making it optional to handle them. Example, of checked exceptions include.

NullPointerException
ArrayIndexOutOfBoundsException
ArithmeticException
IllegalArgumentException, etc

What is an Error?

Errors represent exceptional conditions that are not expected to be caught under normal circumstances. They are typically caused by issues outside the control of applications such as system failures or resource exhaustion. Errors are not meant to be caught or handled by application code. Examples of errors include:

OutOfMemoryError
StackOverflowError
NoClassDefFoundError, etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The. following table describe each:

Keyword	Description
try	The 'try' keyword is used to specify a block where we should place an exception code. It means we cannot use 'try' block alone. The 'try' block must be followed by either catch or finally.
catch	The 'catch' block is used to handle the exception. It. must be preceded by 'try' block, which means we cannot use 'catch' block alone. It can be followed by finally blocked later.
throw	The "throw" keyword is used to throw an exception.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throws	The "throws" keyword is used to declare exceptions it. specifies that there may occur an exception in the method It does not throw an exception. It is always used with method signature.

The try-catch Block

One of the primary mechanism for handling exception in Java is the try catch block. The tried block contains the code that may throw an exception, and the catch block is used to handle the exception effect occurs.

Syntax:

```
try{
    //Code that may throw an exception
}
catch(ExceptionType e){
    //Exception handling code
}
```

Handling Multiple Exceptions

You can handle multiple types of exceptions by providing multiple catch blocks. Each catching a different types of exception. This allows you to tailor your exception handling logic based on the specific type of exception thrown.

Syntax:

```
try{
    //Code that may throw an exception
}
catch(ExceptionType1 e){
    //Exception handling code
}
catch(ExceptionType2 e){
    //Exception handling code
}
...
```

```
    catch(ExceptionTypeN e){
        //Exception handling code
    }
```

The finally Block

In addition to try and catch Java also provides **finally** block which allows you to create cleanup code such as closing resources regardless of whether an exception occurs or not the **finally** block is typically used to release the resources that were acquired in the try block.

Syntax:

```
try{
    //Code that may throw an exception
}
catch(ExceptionType e){
    //Exception handling code
}
finally{
    //Cleanup Code
}
```

Common Scenarios of Java exception

A scenario where NullPointerException occurs

If we have a null value in any variable performing any operation on the variable throws a **NullPointerException**.

```
11 public class NullPointerExceptionDemo
12 {
13     public static void main(String args[]){
14         Scanner sc = new Scanner(System.in);
15         System.out.print("\f");
16         String name = null;
17         System.out.print("\n" + name.length());
18     }
19 }
20
```

```
java.lang.NullPointerException:
null
```

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class NullPointerExceptionDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class NullPointerExceptionDemo
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      try{
         String name = null;
         System.out.print("\n" + name.length());
      }
```

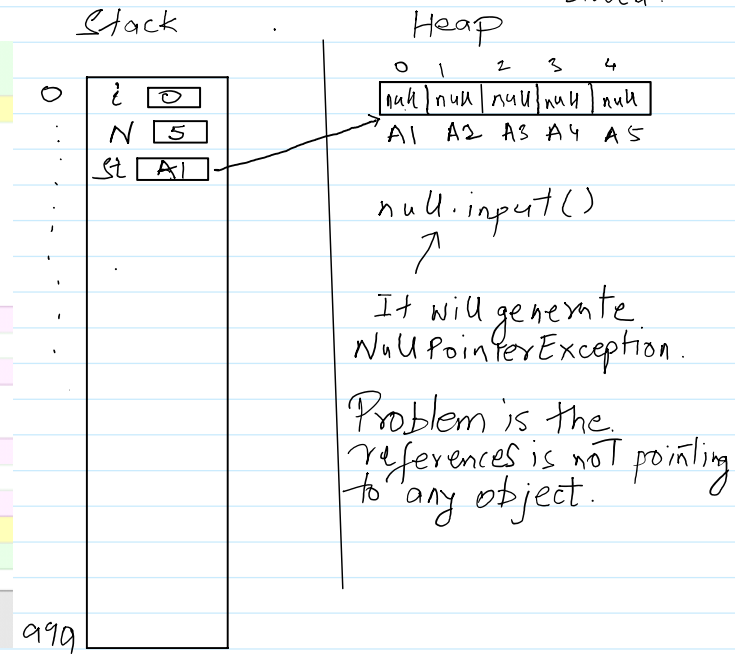
```
    catch(NullPointerException e){
      System.out.print("\n"+e.getMessage());
    }
  }
}
```

Output:

```
BlueJ: Terminal Window - JavaSem2
Options

Cannot invoke "String.length()" because "name" is null
```

3. It keeps the state of variable during method call.
1. Stores primitive type variables.
2. Keep the info of reference types.

all objects or reference types stored.



null.input()

It will generate NullPointerException.

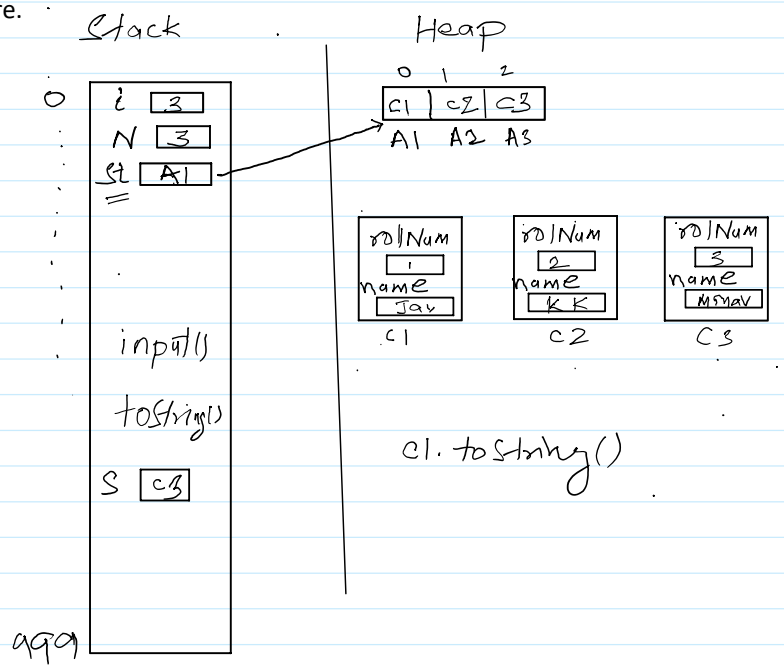
Problem is the references is not pointing to any object.

```
10 public class NullPointerExceptionDemo
11 {
12     public static void main(String args[]){
13         Scanner sc = new Scanner(System.in);
14         System.out.print("\f");
15         int i, N;
16         System.out.print("Enter number of Students: ");
17         N = sc.nextInt();
18         Student st[] = new Student[N];
19
20         for(i = 0; i < N; i++){
21             st[i].input();
22         }
23
24         System.out.print("\nInformation about the students: ");
25         for(Student s : st){
26             System.out.print("\n" + s);
27         }
28     }
29 }
30
```

```
java.lang.NullPointerException:
null
```

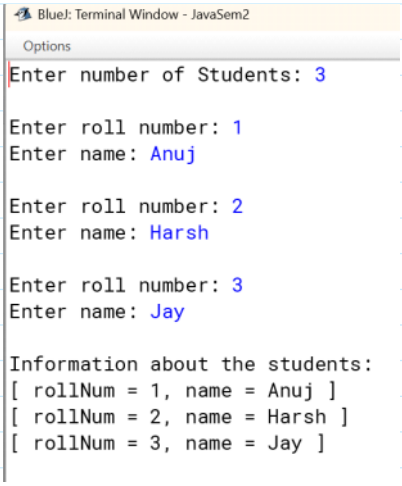
```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class Student here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Student
{
   private int rollNum;
   private String name;

   public Student(int rn, String nm){ //parametrised constructor
      this.rollNum = rn;
      this.name = nm;
   }

   public Student(){} //Non-parametrized constructor

   public void input(){ //user input function
      Scanner sc = new Scanner(System.in);
```

```
      System.out.print("\nEnter roll number: ");
      rollNum = sc.nextInt();
      sc.nextLine(); //to clear the keyboard buffer.
      System.out.print("Enter name: ");
      name = sc.nextLine();
   }

   @Override
   public String toString(){
      return "[ rollNum = " + rollNum + ", name = " + name + " ]";
   }
}

package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class NullPointerExceptionDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class NullPointerExceptionDemo
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");

      int i, N;

      System.out.print("Enter number of Students: ");
      N = sc.nextInt();

      Student st[] = new Student[N];
      try{
         for(i = 0; i < N; i++){
            st[i] = new Student(); //now object is created.
            st[i].input();
         }

      System.out.print("\nInformation about the students: ");
         for(Student s : st){
            System.out.print("\n" + s);
         }
      }
      catch(NullPointerException npe){
         System.out.print("\n"+npe.getMessage());
      }
   }
}
```



toString() is called automatically & it will return string which contain information about s object.

Output:

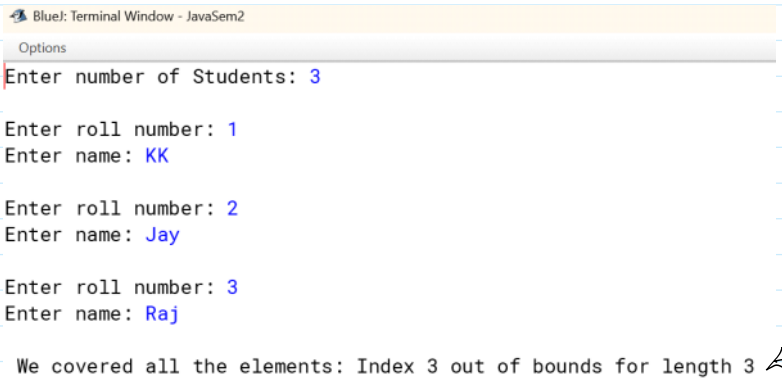


A scenario where ArrayIndexOutOfBoundsException occurs

When we try to access array with invalid index value or subscript value the **ArrayIndexOutOfBoundsException** occurs.

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class NullPointerExceptionDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class NullPointerExceptionDemo
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      int i, N;
      System.out.print("Enter number of Students: ");
      N = sc.nextInt();
      Student st[] = new Student[N];
      try{
         for(i = 0; i <= N; i++){
            st[i] = new Student(); //now object is created.
            st[i].input();
         }

         System.out.print("\nInformation about the students: ");
         for(Student s : st){
            System.out.print("\n" + s);
         }
      }
      catch(NullPointerException npe){
         System.out.print("\n"+npe.getMessage());
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.print("\n We covered all the elements: "+ e.getMessage());
      }
   }
}
```

Output:



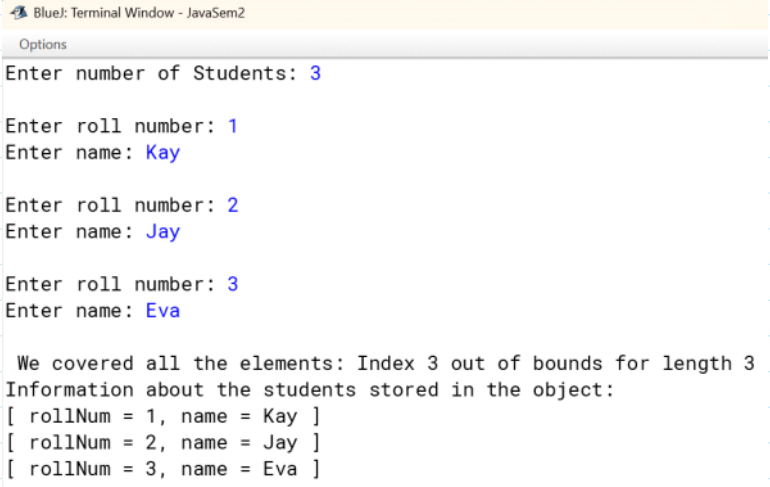
← bravefull exit.

What if we want to print the information stored till the exception has occurred.

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class Student here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Student
{
   private int rollNum;
   private String name;

   public Student(int rn, String nm){ //parametrised constructor
      this.rollNum = rn;
      this.name = nm;
   }

   public Student(){} //Non-parametrized constructor

   public void input(){ //user input function
      Scanner sc = new Scanner(System.in);
      System.out.print("\nEnter roll number: ");
      rollNum = sc.nextInt();
      sc.nextLine(); //to clear the keyboard buffer.
      System.out.print("Enter name: ");
      name = sc.nextLine();
   }

   @Override
   public String toString(){
      return "[ rollNum = " + rollNum + ", name = " + name + " ]";
   }
}

package ExceptionHandling;
import java.util.*;

/**
```

```
 * Write a description of class NullPointerExceptionDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class NullPointerExceptionDemo
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      int i, N;
      System.out.print("Enter number of Students: ");
      N = sc.nextInt();
      Student st[] = new Student[N];
      try{
         for(i = 0; i <= N; i++){
            st[i] = new Student(); //now object is created.
            st[i].input();
         }
      }
      catch(NullPointerException npe){
         System.out.print("\n"+npe.getMessage());
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.print("\n We covered all the elements: "+ e.getMessage());
      }
      finally{
         System.out.print("\nInformation about the students stored in the object: ");
         for(Student s : st){
            System.out.print("\n" + s);
         }
      }
   }
}
```

Output:



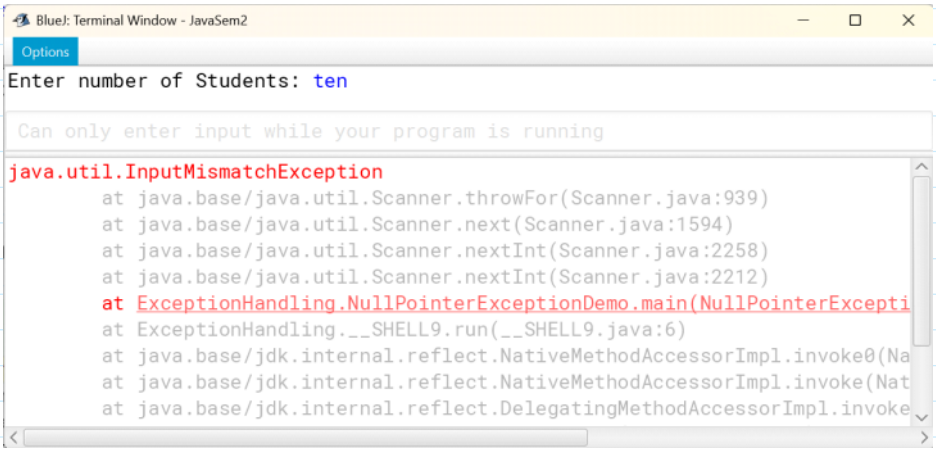
A scenario where InputMismatchException occurs

When we input wrong data in the given variable, then JVM generates InputMismatchException.

```
10 public class NullPointerExceptionDemo
11 {
12     public static void main(String args[]){
13         Scanner sc = new Scanner(System.in);
14         System.out.print("\f");
15         int i, N;
16         System.out.print("Enter number of Students: ");
17         N = sc.nextInt();
18         Student st[] = new Student[N];
19         try{
20             for(i = 0; i <= N; i++){
21                 st[i] = new Student(); //now object is created.
22                 st[i].input();
23             }
24         }
25         catch(NullPointerException npe){
26             System.out.print("\n"+npe.getMessage());
```

```
java.util.InputMismatchException:
null (in java.util.Scanner)
```

Output:



Exception handling code:

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class NullPointerExceptionDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class NullPointerExceptionDemo
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      int i, N;
      while(true){
         try{
            System.out.print("Enter number of Students: ");
            N = sc.nextInt();
            break;
         }
         catch(InputMismatchException e){
            //System.out.print("\n" + e.getMessage());
```

```
            System.out.print("\nPlease enter positive integer value.\n");
            sc.nextLine();
         }
      }
      Student st[] = new Student[N];
      try{
         for(i = 0; i <= N; i++){
            st[i] = new Student(); //now object is created.
            st[i].input();
         }
      }
      catch(NullPointerException npe){
         System.out.print("\n"+npe.getMessage());
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.print("\n We covered all the elements: "+ e.getMessage());
      }
      finally{
         System.out.print("\nInformation about the students stored in the object: ");
         for(Student s : st){
            System.out.print("\n" + s);
         }
      }
   }
}
```

Output:

```
Enter number of Students: 3

Enter roll number: two

null
Insert your data for the object again.

Enter roll number: four

null
Insert your data for the object again.

Enter roll number: 1
Enter name: hay

Enter roll number: 2
Enter name: Jay

Enter roll number: three

null
Insert your data for the object again.

Enter roll number: 3
Enter name: Ian

 We covered all the elements: Index 3 out of bounds for length 3
Information about the students stored in the object:
[ rollNum = 1, name = hay ]
[ rollNum = 2, name = Jay ]
[ rollNum = 3, name = Ian ]
```

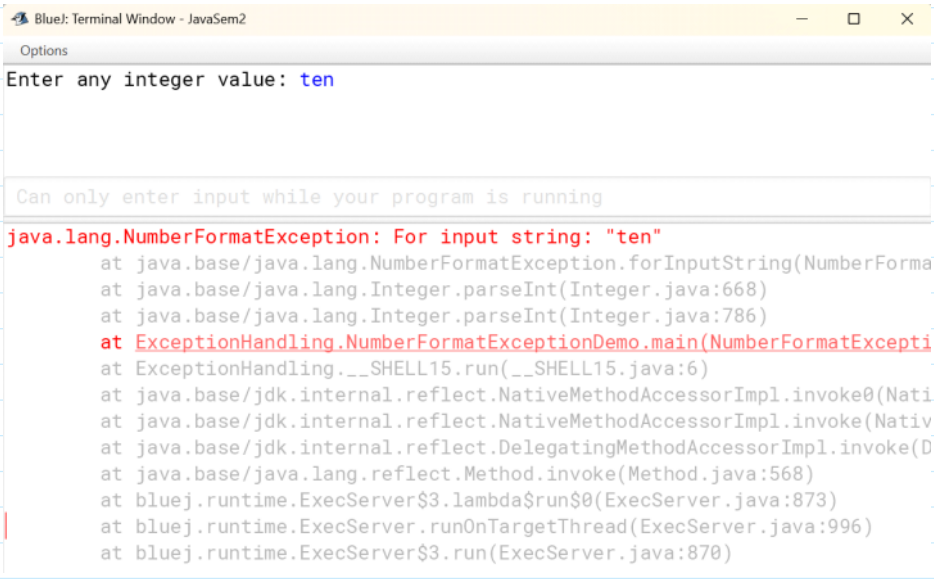
A scenario where NumberFormatExceptiion occurs

If the formatting of any variable or number is mismatched, it may result in to **NumberFormatException**. Suppose we have a string variable that has characters converting this variable into digit will causeway number format exception.

```
11 public class NumberFormatExceptionDemo
12 {
13     public static void main(String args[]){
14         Scanner sc = new Scanner(System.in);
15         System.out.print("\f");
16         String num;
17         System.out.print("Enter any integer value: ");
18         num = sc.next();
19         int x = Integer.parseInt(num);
20         System.out.print("\n x = " + x);
21     }
22 }
23
```

```
java.lang.NumberFormatException:
For input string: "ten" (in java.lang.NumberFormatException)
```

Output:



Exception Handling code

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class NumberFormatExceptionDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class NumberFormatExceptionDemo
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      String num;
      while(true){
```

```
            try{
               System.out.print("Enter any integer value: ");
               num = sc.next();
               int x = Integer.parseInt(num);
               System.out.print("\n x = " + x);
               break; //break the while loop
            }
            catch(NumberFormatException e){
               System.out.print("\n" + e.getMessage());
               System.out.print("\nEnter only integer values.\n");
            }
         }
      }
   }
```

Output:

```
BlueJ: Terminal Window - JavaSem2
Options
Enter any integer value: ten

For input string: "ten"
Enter only integer values.
Enter any integer value: 12.5

For input string: "12.5"
Enter only integer values.
Enter any integer value: 12L

For input string: "12L"
Enter only integer values.
Enter any integer value: 55

 x = 55
```

Conclusion: Exception handling is an essential aspect of Java programming. By following the best practices and adopting effective exception handling strategies, you can write more robust and maintainable Java applications. Remember to catch specific exceptions, keep exception handling simple. log exceptions for debugging purposes, through exceptions appropriately and use custom exceptions when needed. Mastering exception handling will help you to build more reliable software that gracefully handles errors and exceptions.

Homework:

Sections 12.2–12.9

***12.1** (*NumberFormatException*) Listing 7.9, Calculator.java, is a simple command-line calculator. Note the program terminates if any operand is nonnumeric. Write a program with an exception handler that deals with nonnumeric operands; then write another program without using an exception handler to achieve the same objective. Your program should display a message that informs the user of the wrong operand type before exiting (see Figure 12.12).

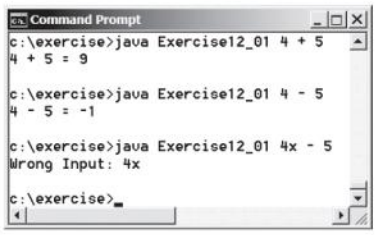


FIGURE 12.12 The program performs arithmetic operations and detects input errors.

***12.2** (*ArrayIndexOutOfBoundsException*) Using the two arrays shown below, write a program that prompts the user to enter an integer between 1 and 12 and then displays the months and its number of days corresponding to the integer entered. Your program should display "wrong number" if the user enters a wrong number by catching `ArrayIndexOutOfBoundsException`.

```
String[] months = {"January", "February", "March", "April",
"May", "June","July", "August", "September", "October",
"November", "December"};
int[] dom = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

***12.3** (*InputMismatchException*) The previous program works well as long as the user enters an integer. Otherwise, you may get another kind of exception. For instance, if you use nextInt() of Scanner, you could have an `InputMismatchException`. Modify it to prevent users entering anything other than an integer.

***12.4** (*IllegalArgumentException*) Modify the `Loan` class in Listing 10.2 to throw `IllegalArgumentException` if the loan amount, interest rate, or number of years is less than or equal to zero.

***12.5** (*IllegalTriangleException*) Programming Exercise 11.1 defined the `Triangle` class with three sides. In a triangle, the sum of any two sides is greater than the other side. The `Triangle` class must adhere to this rule. Create the `IllegalTriangleException` class, and modify the constructor of the `Triangle` class to throw an `IllegalTriangleException` object if a triangle is created with sides that violate the rule, as follows:

```
/** Construct a triangle with the specified sides */
public Triangle(double side1, double side2, double side3)
  throws IllegalTriangleException {
  // Implement it
}
```

Listing 7.9

LISTING 7.9 Calculator.java

```
1  public class Calculator {
2    /** Main method */
3    public static void main(String[] args) {
4      // Check number of strings passed
5      if (args.length != 3) {
6        System.out.println(
7          "Usage: java Calculator operand1 operator operand2");
8        System.exit(1);
9      }
10
11     // The result of the operation
12     int result = 0;
13
14     // Determine the operator
15     switch (args[1].charAt(0)) {
16       case '+': result = Integer.parseInt(args[0]) +
17                          Integer.parseInt(args[2]);
18                 break;
19       case '-': result = Integer.parseInt(args[0]) -
20                          Integer.parseInt(args[2]);
21                 break;
22       case '.': result = Integer.parseInt(args[0]) *
23                          Integer.parseInt(args[2]);
24                 break;
25       case '/': result = Integer.parseInt(args[0]) /
26                          Integer.parseInt(args[2]);
27     }
28
29     // Display result
30     System.out.println(args[0] + ' ' + args[1] + ' ' + args[2]
31       + " = " + result);
32   }
33 }
```

Listing 10.2

LISTING 10.2 Loan.java

```
1  public class Loan {
2    private double annualInterestRate;
3    private int numberOfYears;
4    private double loanAmount;
5    private java.util.Date loanDate;
6
7    /** Default constructor */
8    public Loan() {
9      this(2.5, 1, 1000);
10   }
11
```

```
12    /** Construct a loan with specified annual interest rate,
13        number of years, and loan amount
14      */
15    public Loan(double annualInterestRate, int numberOfYears,
16        double loanAmount) {
17      this.annualInterestRate = annualInterestRate;
18      this.numberOfYears = numberOfYears;
19      this.loanAmount = loanAmount;
20      loanDate = new java.util.Date();
21    }
22
23    /** Return annualInterestRate */
24    public double getAnnualInterestRate() {
25      return annualInterestRate;
26    }
27
28    /** Set a new annualInterestRate */
29    public void setAnnualInterestRate(double annualInterestRate) {
30      this.annualInterestRate = annualInterestRate;
31    }
32
33    /** Return numberOfYears */
34    public int getNumberOfYears() {
35      return numberOfYears;
36    }
37
38    /** Set a new numberOfYears */
39    public void setNumberOfYears(int numberOfYears) {
40      this.numberOfYears = numberOfYears;
41    }
42
43    /** Return loanAmount */
44    public double getLoanAmount() {
45      return loanAmount;
46    }
47
48    /** Set a new loanAmount */
49    public void setLoanAmount(double loanAmount) {
50      this.loanAmount = loanAmount;
51    }
52
53    /** Find monthly payment */
54    public double getMonthlyPayment() {
55      double monthlyInterestRate = annualInterestRate / 1200;
56      double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
57        (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
58      return monthlyPayment;
59    }
60
61    /** Find total payment */
62    public double getTotalPayment() {
63      double totalPayment = getMonthlyPayment() * numberOfYears * 12;
64      return totalPayment;
65    }
66
67    /** Return loan date */
68    public java.util.Date getLoanDate() {
69      return loanDate;
70    }
71  }
```

constructor

Java Throw Exception

24 June 2025 20:59

In Java, the **throw** keyword is used to explicitly throw an exception within a method or block of code. It allows developers to signal that an error has occurred and needs to be handled. The throw exception must be an instance of **Throwable**, or one of its subclasses, such as **Exception** or **RuntimeException**.

Java throw Keyword

The Java throw keyword is used to throw an exception explicitly from a method or block of code. It can be used to throw both checked and unchecked exceptions. It is mainly used to throw a custom exception. It provides developer with a way to trigger an exception whenever a particular undesirable situation occurs manually.

We specify the exception object that is to be thrown. The exception has a message with it that provides the error description. These exceptions may be related to the user input, server hardware malfunctions, invalid computations or many other conditions, depending on the program's requirements.

We can also define our own set of conditions and throw an exception explicitly using the **throw** keyword. For example, we can throw an ArithmeticException if we divide a number by another number. Here we just need to set the condition and throw an exception using the throw keyword.

Syntax:

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

When instance refers to an object of a class that must extend throwable. (which includes the exception class and its subclasses).

```
throw new IOException("Sorry! Device error");
```

Here:

- new is used to create an instance(object) of the exception class.
- The error message provides context about the reason for the exception.

When the instance must be of type throwable or a subclass of throwable, for example, exception is a subclass of Throwable and the user defined exceptions usually extend the exception class.

Key requirements

- The object we throw must be of type Throwable or one of its subclasses.
- For example, Exception is a subclass of Throwable.
- User-defined exception (also known as custom exceptions) usually extend the Exception class.
- If you throw a checked exception and do not handle it inside the method body, you must declare it using the **throws** keyword in the method signature.

Why use throw in Java?

Manually throwing exceptions allows programmers to:

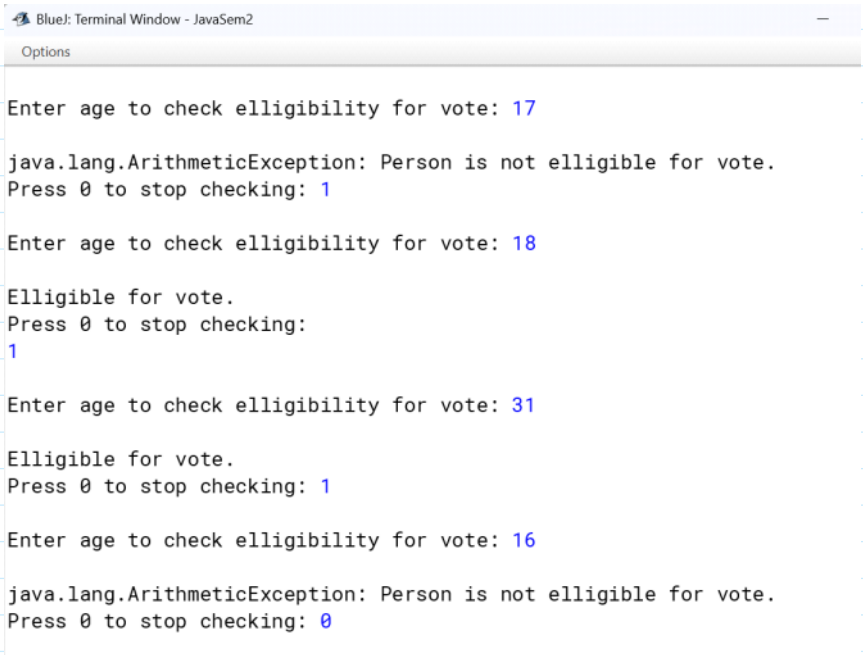
- Validate Inputs (for example, throwing an exception if the user input is invalid).
- Handle specific conditions (For example, throwing exceptions if a requested resource is not found).
- Create more meaningful error messages that are specific to the application's context.
- Improved debugging by providing custom error information.

Java throw Keyword Example:

```
package ExceptionHandling;
import java.util.*;

/**
 * Write a description of class ThrowException here.
 *
```

```
 * @author (your name)
 * @version (a version number or a date)
 */
public class ThrowException
{
   public static void validate(int age){
      if(age < 18){
         throw new ArithmeticException("Person is not elligible for vote.");
      }
      else{
         System.out.print("\nElligible for vote.");
      }
   }

   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      //int ch;
      while(true){
         try{
            System.out.print("\nEnter age to check elligibility for vote: ");
            validate(sc.nextInt());
         }
         catch(ArithmeticException ae){
            System.out.print("\n"+ae);
         }
         System.out.print("\nPress 0 to stop checking: ");
         if(sc.nextInt() == 0){
            break;
         }
      }
   }
}
```

Output:



Example: Throwing a Checked Exception

In Java checked exceptions are the exceptions that the compiler requires you to either handle using a try-catch block or declare using **throws** keyword in the method signature. These exceptions typically arise from external sources such as file operations, network connections or database access operations that are prone to failure.

Unlike checked exceptions (which are subclasses of RuntimeException) Check the exceptions are subclasses of Exception (but not of RuntimeException).

```
package ExceptionHandling;
import java.util.*;
import java.io.*;


/**
 * Write a description of class CheckedException here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class CheckedException
{
   public static void method()throws FileNotFoundException{
      FileReader fr = new FileReader("GeneratingException.java");
      BufferedReader br = new BufferedReader(fr);
      //Explicitly throwing a checked exception
      throw new FileNotFoundException();
   }
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");

      try
      {
         method();
      }
      catch (FileNotFoundException fnfe)
      {
         fnfe.printStackTrace();
      }
      System.out.print("\nRest of the code...");
   }
}
```

Output:

```
Rest of the code...
```

```
Can only enter input while your program is running
```

```
java.io.FileNotFoundException: GeneratingException.java (The system cannot find the file specified)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:216)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:111)
        at java.base/java.io.FileReader.<init>(FileReader.java:60)
        at ExceptionHandling.CheckedException.method(CheckedException.java:15)
        at ExceptionHandling.CheckedException.main(CheckedException.java:26)
        at ExceptionHandling.__SHELL6.run(__SHELL6.java:6)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.jav
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessor
        at java.base/java.lang.reflect.Method.invoke(Method.java:568)
        at bluej.runtime.ExecServer$3.lambda$run$0(ExecServer.java:873)
        at bluej.runtime.ExecServer.runOnTargetThread(ExecServer.java:996)
        at bluej.runtime.ExecServer$3.run(ExecServer.java:870)
```

Throwing User-defined Exception

The exception is everything else under the **Throwable** class. Java allows developers to create their own custom exceptions by extending the **Exception** class or one of its subclasses. They are typically used when you want to enforce application specific rules and constraints that are not covered by Java's built-in exceptions.

In this example, we will define a custom exception called UserDefinedException and throw it manually using the throw keyword. We will also catch and handle it using a try-catch block, just like we would with any other checked exceptions.

```
public class UserDefinedException extends Exception
{
   public UserDefinedException(String msg){
      //Callling the constructor of super class or parent class
      super(msg);
   }
}
```

↑ sending 'msg' to Exception class which is a parent.

```
import java.util.*;
public class TestUserDefinedException
{
   public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
      System.out.print("\f");
      try{
         throw new UserDefinedException("This is user defined exception.");
      }
      catch(UserDefinedException ude){
         System.out.print("\nCaught the Exception");
         System.out.print("\n"+ude.getMessage());
      }
   }
}
```

This is a constructor Message.

Throwing the object created by new keyword

Catching the object.

OUTPUT:

```
BlueJ: Terminal Window - JavaSem2
Options

Caught the Exception
This is user defined exception.
```

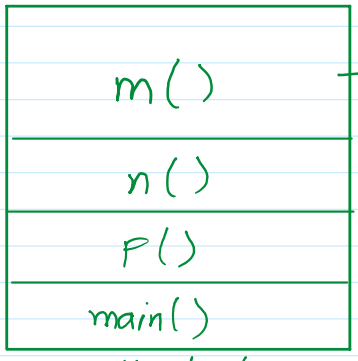
Exception Propagation in Java

An exception is a first thrown from the top of the stack, and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Note: By default, unchecked exceptions are forward in calling chain (propagated).

```
package ExceptionHandling;


/**
 * Write a description of class ExceptionPropagating here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class ExceptionPropagation
{
   public void m(){
      int data = 100/0;
   }

   public void n(){
      m();
   }

   public void p(){
      try{
         n();
      }
      catch(Exception e){
         System.out.print("\nException handled");
      }
   }

   public static void main(String args[]){
      System.out.print("\f");
      ExceptionPropagation obj = new ExceptionPropagation();
      obj.p();
      System.out.print("\nNormal flow...");
   }
}
```

exception not handled, so it is propagated to the previous method. n(), where it is not handled, again it is propagated to p() method. where exception is handled.

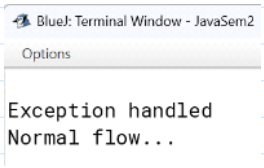


exception occured.

call stack.

Exception can be handled in any method in call stack either in main(), p(), n() or m() method.

Output:


BlueJ: Terminal Window - JavaSem2
Options
Exception handled
Normal flow...