60.1.1. What are exceptions?

01:54

An exception by its very definition is something which is not normal.

In programming, if the program does not run as expected due to some abnormal event or situation, the error produced is called an exception.

Java provides easy to use constructs to handle such situations.

Even before we learn how to handle exceptions let us see an exception and try fixing it.

The below example code when run produces an exception called java.lang.ArithmeticException.

Click on **Submit** button to see the exception details as given below.

```
Caused by: java.lang.ArithmeticException: / by zero   // it has exception class name followed by error message
        at ExceptionDemo1.main(ExceptionDemo1.java:5) // it has the class/method name and the line number which tr
        ... 6 more
```

After you click on `Submit` and see the exception message, replace the value `0` of the `divisor` variable with `2` to fix the problem.

Important: Please note that the blue animating arrow which is shown when you click **Submit**, is shown by our **intelligent error detection system**. When you start coding using a regular IDE during work, you will not be helped like this. You will only be provided the error information (stack trace) without this animating arrow. Hence, learning how to read and understand exception stack traces becomes an essential part of programming.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Exceptio...

```
1  package q11317;
2  public class ExceptionDemo1 {
3      public static void main(String[] args) {
4          int number = 34;
5          int divisor = 2;
6          int result = number / divisor;
7          System.out.println("result = " + result);
8      }
9  }
```

Terminal Test cases

< Prev Reset Submit Next

60.1.2. How to read and understand exception stack trace?

01:24

When an exception occurs while the code is running, the JVM tries to provide as much information as possible regarding the line of code which triggered the exception.

This information usually spans multiple lines which is called the exception stack trace.

Click on **Submit** button to see the exception stack trace produced while executing the code.

It is very easy to fix the error in this scenario. However, we will first click on the `Submit` button and then learn to read the stack trace information provided.

```
Caused by: java.lang.ArithmeticException: / by zero     // it contains the exception class name followed by message
        at ExceptionDemo2.divide(ExceptionDemo2.java:9)   // line - 9
        at ExceptionDemo2.main(ExceptionDemo2.java:5)     // line - 5
        ... 6 more
```

The exception stack trace contains two lines from the ExceptionDemo2 class.

The `top most line` in the **exception stack trace** which is from our code (meaning from a class written by us) is responsible for causing the exception.

When the statement in **line 9** in our code (ExceptionDemo2 class) is analyzed for the **ArithmeticException** with error message `/ by zero` , we can easily figure out that the value contained in the variable `divisor` is `0` (zero).

There can be some scenarios where the `top most line` is from a class available in Java standard classes or a class written by a third party library provider, in such cases we will have to ignore such lines till we find the lines which are from classes written by us.

After you have understood the above stated rules, correct the error in the code by replacing the value `0` of variable `number2` with `2` and click on **Submit** .

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Exceptio...

```
1   package q11318;
2   public class ExceptionDemo2 {
3       public static void main(String[] args) {
4           int number1 = 34;
5           int number2 = 2;
6           int result = divide(number1, number2);
7           System.out.println("result = " + result);
8       }
9       public static int divide(int number, int divisor) {
10          return number / divisor;
11      }
12  }
```

Terminal Test cases

‹ Prev Reset Submit Next

60.1.3. How to read and understand exception stack trace?

03:01

The exception stack trace can include classes which are not written by us. The below code when submitted will terminate with an exception whose stack trace will contain method call information from Java classes in java.lang package.

As a learner it is easy to get lost when we see so many error lines.

However, it is extremely easy to pinpoint the line with error if we follow a simple thumb rule.

To start with let us click on `Submit` button to see the exception stack trace produced by the code.

Do not try to fix the code. First let us try to understand the exception stack trace we get when we click `Submit`.

```
Caused by: java.lang.NumberFormatException: For input string: "4a"  // notice the exception class name and error messa
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at ExceptionDemo3.convertAndAdd(ExceptionDemo3.java:10)
        at ExceptionDemo3.main(ExceptionDemo3.java:5)
        ... 6 more
```

The first thing we should read in the exception stack trace is the line starting with `Caused by:` which contains the **name of the exception class** and the **error message**.

In our case the name of the exception class is NumberFormatException

And the error message is `For input string: "4a"`

It is always a good practice to read about the exception class. [Hint: Click on the name of the exception class to read about it.]

The exception stack trace in total contains **5 lines**. Out of which **2 lines** are from our class **ExceptionDemo3** and the remaining **3 lines** are from classes in Java's standard library.

Our goal is to find out which line in our code is responsible for triggering the exception. For all practical purposes we can safely assume that classes in Java code are not responsible for this exception.

Which means **we can safely ignore all the top 3 lines** in the stack trace which are from classes `java.lang.NumberFormatException` and `java.lang.Integer`.

As mentioned earlier we should scan from top to bottom to find the `top most line` in the **exception stack trace** which is from our code (meaning from a class written by us, which in our case is **ExceptionDemo3**).

We will notice that **line 10** is what we are looking for.

Sample Test Cases +

Explorer

Exceptio...

```
1  package q11319;
2  public class ExceptionDemo3 {
3      public static void main(String[] args) {
4          String text1 = "3";
5          String text2 = "4";
6          int result = convertAndAdd(text1, text2);
7          System.out.println("result = " + result);
8      }
9      public static int convertAndAdd(String number1Text, String number2Text) {
10         int number1 = Integer.parseInt(number1Text);
11         int number2 = Integer.parseInt(number2Text);
12         return number1 + number2;
13     }
14 }
```

Terminal Test cases

‹ Prev Reset Submit Next

Home Learn Anywhere

60.1.4. Fix the problem

00:54

Click `Submit` to find out the problem.

The code should actually print the **fourth** element passed in the arguments to the main method.

[**Hint:** Click on the **animating blue arrow** in the exception stack trace to understand the error.]

Note: Please don't change the package name.

Sample Test Cases

Explorer

Exceptio...

```
1  package q11320;
2  public class ExceptionDemo4 {
3      public static void main(String ... args) {
4          System.out.println(args[3]);
5      }
6  }
```

Terminal Test cases

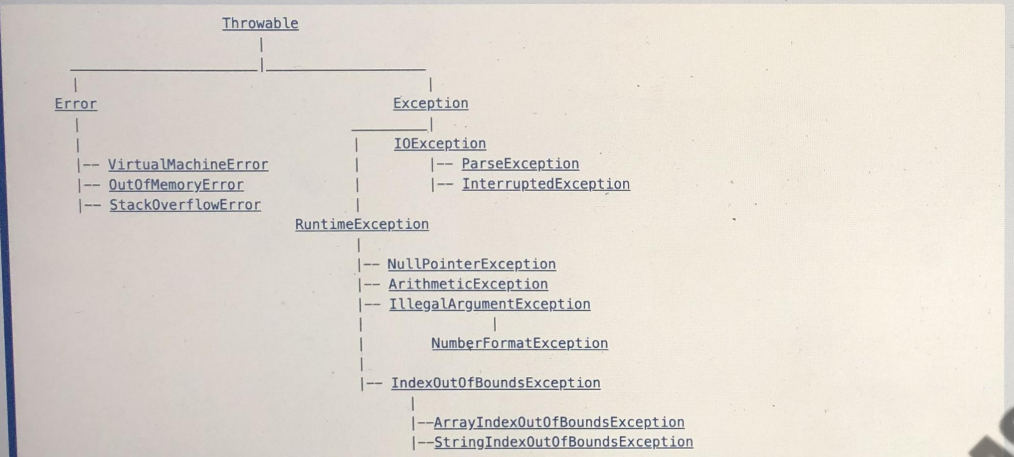
< Prev Reset Submit Next

61.1.1. Understanding exception class hierarchy

05:39

In Java, Throwable is the super class for all Error classes and Exception classes.

Among hundreds of exception and error classes, below are few commonly used ones and their **exception hierarchy**:



In the above hierarchy, `Throwable` and all subclasses of Throwable that **are not** subclasses of either `RuntimeException` or `Error` are called checked exceptions.

All the other subclasses of `Error` and `RuntimeException` are called unchecked exceptions.

The Throwable class stores the method call stack trace and also the error message that is printed when an exception occurs.

Even though we have classes called `Error` and `Exception` as the main subclasses of `Throwable`, the whole concept of handling the exception scenarios is called exception handling. We do not say Throwable-handling, Error-handling and Exception-handling separately based on their class names. In general programming terminology, the concepts along with the constructs which are employed to handle erroneous or abnormal situations is called `exception handling`.

Select all the correct statements given below.

- [ ] ParseException is an unchecked exception.
- [ ] NumberFormatException is an checked exception.
- [x] RuntimeException is an unchecked exception.
- [x] NullPointerException is an unchecked exception.
- [x] ArithmeticException is an unchecked exception.
- [ ] StackOverflowError is a checked exception.

Home Learn Anywhere

62.1.1. Understanding checked and unchecked exceptions

03:05

An exception is thrown to signal an error condition.

Java provides a way to catch the exceptions and handle them by either taking corrective course of action or notifying the user or doing what ever is needed.

This handling is done using a `try-catch` block or propagating the exception to the caller.

The compiler flags an error if we invoke methods which throw checked exceptions and do not handle them.

For example, the static method Thread.sleep(long milliSeconds) in Thread class throws an InterruptedException, which is of type checked exception. Below code demonstrates the usage of `try-catch` block. Notice how the statement containing the call to the sleep method is wrapped in the `try-catch` block.

```
try {  // try-catch block start
    Thread.sleep(2000);
} catch (InterruptedException e) {  // catch clause
    e.printStackTrace();
}  // try-catch block end
```

Click on `Submit` to see the error.

To fix the code, write the try-catch block appropriately.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

TryCatch...

```
1  package q11322;
2  public class TryCatchDemo {
3      public static void main(String[] args) {
4          System.out.println("Before sleep...");
5          try{
6          Thread.sleep(2000);
7          }catch(InterruptedException e){
8              e.printStackTrace();
9          }
10         System.out.println("After sleep...");
11     }
12 }
```


Terminal Test cases

‹ Prev Reset Submit Next ›

62.1.2. Checked and Unchecked Exceptions

02:39

Checked Exceptions:

Checked exceptions are exceptions that are checked at compile-time. This means that the compiler checks whether the exceptions are handled or declared in the method signature using the `throws` keyword. If a checked exception is not handled or declared, the code will not compile.

Examples of checked exceptions include:

- `IOException` (and its subclasses like `FileNotFoundException`)
- `ClassNotFoundException`
- `SQLException`
- `InterruptedException`

Checked exceptions generally represent external conditions that are outside the control of the program, such as I/O errors, network errors, or database connectivity issues.

Unchecked Exceptions:

Unchecked exceptions are exceptions that are not checked at compile-time. These exceptions are a subclass of the `RuntimeException` class or one of its subclasses. The compiler does not enforce handling or declaring unchecked exceptions, as they are assumed to be caused by programming errors that should be detected and fixed during development.

Examples of unchecked exceptions include:

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`
- `IllegalArgumentException`
- `ClassCastException`

Unchecked exceptions usually represent internal conditions that are within the control of the program, such as invalid method arguments, null references, or array index violations.

Handling Exceptions:

Checked exceptions must be either caught using a `try-catch` block or declared in the method signature using the `throws` keyword. This is because checked exceptions are assumed to be unavoidable and beyond the control of the program.

```
try {
    // Code that might throw a checked exception
} catch (CheckedException e) {
    // Handle the exception
}
```

Unchecked exceptions, on the other hand, do not require explicit handling or declaration, but it is generally considered good practice to handle them appropriately to prevent program termination and provide proper error handling.

```
int[] numbers = {1, 2, 3};
try {
    int value = numbers[5]; // Unchecked ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
    // Handle the exception
}
```


- The program throws an unchecked exception
- (●) The compiler generates an error
- The program compiles successfully
- The program crashes at runtime

‹ Prev | Reset | Submit | Next ›

62.1.3. Understanding StackOverflowError

03:22

StackOverflowError is a subclass of **Error** and hence an unchecked exception. It is thrown by JVM.

While executing code when JVM exhausts the stack space available to store the method call information, it throws this error. `StackOverflowError` is not an instance (subclass) of an `Exception` class. It is a subclass of `Error`.

In the below code, you will notice that the `main` method calls the `someMethod()`. And the `someMethod()` internally calls itself.

The execution starts with the `main`. At this moment the JVM pushes the `main` method details in to a stack called **method call stack**.

Method Call Stack

method-1	main(String[] args)

When the `main` for the first time calls `someMethod()`, JVM pushes the method information of `someMethod()` also into the **method call stack**. At this moment there are two entries in the **method call stack**.

Method Call Stack

method-call-2	someMethod()
method-call-1	main(String[] args)

When the `someMethod()` internally call itself again, JVM pushes the method information of `someMethod()` once again into the **method call stack**. At this moment there are three entries in the **method call stack**.

Method Call Stack

method-call-3	someMethod()
method-call-2	someMethod()
method-call-1	main(String[] args)

Since there is no condition in `someMethod()` which will stop itself from calling itself, it will infinitely keep calling itself. Eventually JVM will exhaust the stack space available to store the method call information for each and every call of `someMethod()`.

It is at this point when JVM gives up, it throws the **StackOverflowError** signally that it is out of stack space and just cannot do anything.

As a programmer whenever we see a **StackOverflowError**, we should analyze our code and remove the infinite recursion.

Subclasses of `Error` usually signal such fatal problems in JVM, which are not recoverable. Meaning we should not try to enclose

Sample Test Cases +

Explorer

StackOve...

```
1  package q35975;
2  public class StackOverflowErrorDemo {
3      private static int counter = 0;
4      public static void main(String... args) {
5          try {
6              someMethod();
7          } catch (Error ste) {
8              System.out.println("Stack overflow occurred");
9          }
10     }
11     public static void someMethod() {
12         counter++;
13         someMethod();
14     }
15 }
```

Terminal Test cases

‹ Prev Reset Submit Next ›

Home Learn Anywhere

[redacted] Suppor

62.1.4. Checked Exception - FileNotFoundException

03:05

What is a FileNotFoundException?

The `FileNotFoundException` is a subclass of the `IOException` class, and it is thrown when an attempt to open a file fails due to various reasons, such as:

- The file doesn't exist in the specified path.
- The application doesn't have permission to access the file.
- The file is already being used by another process.

Handling a FileNotFoundException

Since the `FileNotFoundException` is a checked exception, you must handle it using a `try-catch` block or declare it in the method signature using the `throws` keyword.

You will be learning the file-handling concepts in later lessons.

Your task:

You have been given a partially completed Java program, **FileReaderExample**, which aims to read the contents of a file specified by the user. However, certain parts of the code are missing. Implement error handling to handle the situation where the specified file is not found (**FileNotFoundException**).

Sample Test Cases +

Explorer

FileRead... file1.txt

```
1  import java.io.*;
2
3  public class FileReaderExample {
4      public static void main(String[] args) {
5          BufferedReader reader = new BufferedReader(new InputStreamReader(System.in
6
7          try
8          {
9              // Prompt the user to enter the file name
10             System.out.print("File name: ");
11             String fileName = reader.readLine();
12
13             // Open the file for reading
14             FileReader fileReader = new FileReader(fileName);
15             BufferedReader fileBufferedReader = new BufferedReader(fileReader);
16
17             // Code to read file and display its contents
18             String line;
19             while ((line = fileBufferedReader.readLine()) != null) {
20                 System.out.println(line);
21             }
22
23             // Close the file reader
24             fileBufferedReader.close();
25
26         // Handle FileNotFoundException
27         } catch (FileNotFoundException e    ) {
28
29             System.out.println("File not found");
30
31         } catch (IOException e) {
32             // Handle IOException when reading user input or file contents
33             System.err.println("An error occurred: " + e.getMessage());
34         }
35     }
36 }
37
```


Terminal Test cases

‹ Prev Reset Submit Next ›

62.1.5. Unchecked Exception - ArrayIndexOutOfBoundsException

05:53

What is an ArrayIndexOutOfBoundsException?

- The `ArrayIndexOutOfBoundsException` is an unchecked exception in Java that occurs when you try to access an array element with an index that is outside the bounds of the array. It is a subclass of the `IndexOutOfBoundsException` class, which is a subclass of the `RuntimeException` class.

Handling an ArrayIndexOutOfBoundsException

Since the `ArrayIndexOutOfBoundsException` is an unchecked exception (a subclass of `RuntimeException`), you are not required to handle it explicitly with a try-catch block. However, it's generally a good practice to handle or prevent such exceptions to avoid unexpected program termination.

Your task:

You have been provided with a partially completed Java program, `ArrayIndexHandler`, which aims to handle array index validation. Implement array index validation to ensure that the index provided by the user is within the bounds of the array.

- Use a `try-catch` block to catch `ArrayIndexOutOfBoundsException`.
- If the index is valid, print the element at the specified index.
- If the index is invalid, print an error message indicating that the index is out of bounds.

Sample Test Cases +

Explorer

ArrayInde...

```java
import java.util.Scanner;
public class ArrayIndexHandler {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input array
        int size = scanner.nextInt();
        int[] numbers = new int[size];

        for (int i = 0; i < size; i++) {
            numbers[i] = scanner.nextInt();
        }

        // Input index
        int index = scanner.nextInt();

        // Array index validation
        try{
            System.out.println("Element at index "+index+" is: "+numbers[index]);
        }

        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Index out of bounds");
        }
    }
}
```


Terminal Test cases

‹ Prev Reset Submit Next ›

63.1.1. Understanding try-catch-finally syntax

00:56

The syntax for try-catch-finally syntax is as given below:

```
try {
    // try block
} catch (ExceptionName1  referenceName1) {
    // catch block
} catch (ExceptionName2  referenceName2) {
    // another catch block
} finally {
    // finally block
}
```

The block of code which is just after the `try` keyword is called the `try block`.

The block of code which is just after the `catch` keyword is called the `catch block`.

The block of code which is just after the `finally` keyword is called the `finally block`.

There can be a `try block` with only `catch blocks` and without the `finally block`.

Similarly there can be a `try block` without `catch blocks` and with just one `finally block`.

When an exception is thrown, the `try block` does not complete normally, meaning the control flow will jump into the `catch block`. And after the catch block is executed it will later enter into the `finally block` and execute the code.

Click on Live Demo to understand the working of `try`, `catch` and `finally` constructs.

In a try-catch-finally construct, when an exception occurs, which block(s) will execute?

- Only the try block.
- Only the catch block.
- Only the finally block.
- Both the catch and finally blocks.

63.1.2. Understanding try-catch-finally syntax

00:38

The `try` block can have zero or more `catch blocks`. However, a `try block` can have zero or **only one** `finally block`.

Note that if a `try block` has one or more `catch blocks`, then the `finally block` should be written only after the last `catch block`.

The finally block cannot appear before or in between the `catch blocks`.

✔ Valid

```
try {
    // try block
} catch (ExceptionName1 referenceName1) {
    // catch block
} catch (ExceptionName2 referenceName2) {
    // another catch block
} finally {
    // finally block
}
```

✔ Valid

```
try {
    // try block
} catch (ExceptionClassName referenceName) {
    // catch block
}
```

✔ Valid

```
try {
    // try block
} finally {
    // finally block
}
```

Select all the correct usages of `try`, `catch` and `finally` blocks given below:

- [ ]
```
try {
  ...
} finally {
  ...
} catch (ExceptionClassName referenceName) {
  ...
}
```

- [ ]
```
catch {
  ...
} finally {
  ...
} try (ExceptionClassName referenceName) {
  ...
}
```

- [ ]
```
try {
  ...
} finally (ExceptionClassName referenceName) {
  ...
} catch {
  ...
}
```

- [x]
```
try {
  ...
} catch (ExceptionClassName referenceName) {
  ...
} finally {
  ...
}
```

- [x]
```
try {
  ...
} finally {
  ...
}
```

‹ Prev Reset Submit Next ›

63.1.3. Understanding multi-catch clause

02:32

In **Java 7** and later versions, multiple catch blocks can be combined into a single block.

For example the two catch blocks in the below code:

```
try {
    // try block
} catch (ExceptionClassName1  referenceName1) {
    // catch block
} catch (ExceptionClassName2  referenceName2) {
    // another catch block
} finally {
    // finally block
}
```

can be combined into a single catch block as given below:

```
try {
    // try block
} catch (ExceptionClassName1 | ExceptionClassName2  referenceName) {
    // multi-catch block
} finally {
    // finally block
}
```

Note that only Throwable and its subclasses can be caught in the `catch` statement.

It is always a good practice to catch the exceptions and also print their stack trace by calling the printStackTrace() method. The **printStackTrace()** method is present in the superclass Throwable, hence it is available in every exception class.

Note there may be situations when you may not want call the **printStackTrace()** method, however let it be a conscious decision.

The `finally` block is very useful for writing the cleanup code. Since the `finally block` is always executed after the `try block`, any code that is written inside the `finally block` will be executed before the control is transferred either by a **break** statement or a **continue** statement or even by a **return** statement.

In **Java 7** there is a new construct called **try-with-resources**. We will learn more about it later in sections related to streams in **java.io** package.

Select all the correct statements for the below code:

```
public class TrickyExample {
    public static void main(String[] args) {
        String text1 = "3";
        String text2 = "4g";
        System.out.println(getTotal(text1, text2));
    }
    public static int getTotal(String text1, String text2) {
        int value1 = 0;
        int value2 = 0;
        try {
            value1 = Integer.parseInt(text1);
            value2 = Integer.parseInt(text2);
```

- [ ] The code will print `3`.
- [ ] The code will print `34g`.
- [ ] The code will print `-1`.
- [x] The code will print `-2`.

‹ Prev Reset Submit Next ›

Home Learn Anywhere

[redacted]

63.1.4. Write a Java program to handle an ArithmeticException - divided by zero

03:42

Write a Java program to handle an `ArithmeticException` **divide by zero** using exception handling.

Write a class called `Division` with a **main()** method. Assume that the **main()** method will receive two arguments which have to be internally converted to **integers**.

Write code in the **main()** method to divide the first argument by the second (as integers) and print the result (i.e the quotient).

If the command line arguments to the **main()** method are **"12", "3"**, then the program should print the output as:

```
Result = 4
```

If the command line arguments to the **main()** method are **"55", "0"**, then the program should print the output as:

```
Exception caught : divide by zero occurred
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Division.j...

```
1  package q11329;
2  import java.util.*;
3  public class Division{
4      public static void main(String[] args){
5          Scanner sc = new Scanner(System.in);
6          try{
7              int x = Integer.parseInt(args[0]);
8              int y = Integer.parseInt(args[1]);
9              System.out.println("Result = "+(x/y));
10         }catch(ArithmeticException e){
11             System.out.println("Exception caught : divide by zero occurred");
12         }
13     }
14 }
```


Terminal Test cases

‹ Prev Reset Submit Next ›

Home Learn Anywhere

63.1.5. Write a Java program to illustrate Finally block

07:21

Write a Java program to handle an `ArithmeticException` **divided by zero** by using **try**, **catch** and **finally** blocks.

Write the **main()** method with in the class `MyFinallyBlock` which will receive **four** arguments and convert the first two into integers, the last two into float values.

Write the **try**, **catch** and **finally** blocks separately for finding division of two **integers** and two **float** values.

If the input is given as command line arguments to the **main()** as **"10", "4", "10", "4"** then the program should print the output as:

```
Result of integer values division : 2
Inside the 1st finally block
Result of float values division : 2.5
Inside the 2nd finally block
```

If the input is given as command line arguments to the **main()** as **"5", "0", "3.8", "0.0"** then the program should print the output as:

```
Inside the 1st catch block
Inside the 1st finally block
Result of float values division : Infinity
Inside the 2nd finally block
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

MyFinally...

```
1  package q11330;
2  import java.util.*;
3  public class MyFinallyBlock {
4      public static void main(String[] args){
5          Scanner sc = new Scanner(System.in);
6          try{
7              int a = Integer.parseInt(args[0]);
8              int b = Integer.parseInt(args[1]);
9              System.out.println("Result of integer values division : "+(a/b));
10         }catch(ArithmeticException e){
11             System.out.println("Inside the 1st catch block");
12         }
13         finally{
14             System.out.println("Inside the 1st finally block");
15         }
16         try{
17             float x = Float.parseFloat(args[2]);
18             float y = Float.parseFloat(args[3]);
19             System.out.println("Result of float values division : "+(x/y));
20         }finally{
21             System.out.println("Inside the 2nd finally block");
22         }
23     }
24 }
```


Terminal Test cases

< Prev Reset Submit Next >

Home Learn Anywhere

[redacted] Suppor

63.1.6. Write a Java program to illustrate Multiple catch blocks

05:19

Write a Java program to illustrate **multiple catch blocks** in exception handling.

Write a method **multiCatch(int[] arr, int index)** in the class `MultiCatchBlocks` where `arr` contains integer array values and `index` contains an integer value.

Write the code in **try** block to print the value of **arr[index]** and also print the division value of **arr[index]** by **index**.

Write the **catch** blocks for

1. ArithmeticException which will print "**Division by zero exception occurred**"
2. ArrayIndexOutOfBoundsException which will print "**Array index out of bounds exception occurred**".
3. Exception (which catches all exceptions) will print "**Exception occurred**"

Note: Please don't change the package name.

Sample Test Cases +

Explorer

MultiCatc... MultiCatc...

```
1  package q11331;
2  import java.time.*;
3  public class MultiCatchBlocks {
4
5      // Write the code
6      public void multiCatch(int[] arr, int index){
7          try{
8              System.out.println(arr[index]);
9              System.out.println(arr[index]/index);
10         }catch(ArithmeticException e){
11             System.out.println("Division by zero exception occurred");
12         }catch(ArrayIndexOutOfBoundsException e){
13             System.out.println("Array index out of bounds exception occurred");
14         }catch(Exception e){
15             System.out.println("Exception occurred");
16         }
17     }
18 }
```


Terminal Test cases

‹ Prev Reset Submit Next ›

Unit 5 - Lesson 5 - Usage of throw, throws

About this unit

Usage of throw, throws

Usage of throw, throws

Unit • 100% completed

Java - Custom Exceptions - I

Assessment

64.1.1. Understanding throw and throws keywords

04:41

The **throw** keyword is used to write the throw statement which causes the exception to be thrown.

The syntax for the **throw statement** is as below:

```
throw  a ThrowableInstance;
```

For example:

```
throw new Exception("The world is about to end!");
```

As you can notice the `throw` clause should be followed by an instance of Throwable or one of its subclasses.

We can also throw custom exceptions, about which we will learn later.

The `throws` keyword is used in the method or constructor declaration. It is used to inform (or list) all the **checked** exceptions which the method or constructor body can throw during execution.

Note that the `throws` clause in the method or constructor declaration **need not** list the **unchecked** exceptions that are thrown by the code in the method or constructor body. The syntax for the throws is as given below:

```
methodModifiersList  returnType methodName(parameterList) throws  ExceptionClassName1, ExceptionClassName2, ... {
}
```

For example:

```
public void setAge(int age) throws InvalidAgeException {
    if (age < 0 || age > 999) { //assuming super-humans can live 999 years
         throw new InvalidAgeException ("Invalid age. Valid range for age is between 0 and 999.");
    }
    this.age = age;
}
```

The `throw` clause can throw **only a single exception** at a time.

However, the `throws` clause **can specify multiple exceptions** the method or constructor throws.

Note that a method or a constructor which does not want to handle a checked exception can let it go out of it, by including that exception class name in its throws declaration.

For example, in the below code the `Student(String name, int age)` constructor does not handle the `InvalidAgeException` which is thrown by the `setAge` method using a `try-catch` block. Instead the constructor included `InvalidAgeException` in its `throws` clause.

What will be the output when the following code is executed, assuming that all necessary classes are imported?

```
public class ThrowAndThrowsExample {
    public static void main(String[] args) {
        Student st1 = null;
        Student st2 = null;
        try {
```


- Successfully created st1.
  st1: name = Ganga, age = 25
  Could not create st2. Error message is: Invalid age: 1003. Valid range for age is between 0 and 999.
- Could not create st1. Error message is: Invalid age: 25. Valid range for age is between 0 and 999.
  Successfully created st2.
  st2: name = Yamuna, age = 1003
- Could not create st1. Error message is: Invalid age: 25. Valid range for age is between 0 and 999.
  Could not create st2. Error message is: Invalid age: 1003. Valid range for age is between 0 and 999.
- Successfully created st1.
  st1: name = Ganga, age = 25
  Successfully created st2.
  st2: name = Yamuna, age = 1003

Home Learn Anywhere

64.1.2. Write a Java program for creation of illustrating throw

16:07

Write a Java program for creation of illustrating **throw**.

Write a class `ThrowExample` contains a method **checkEligibilty(int age, int weight)** which throws an **ArithmeticException** with a message "**Student is not eligible for registration**" when **age < 12** and **weight < 40**, otherwise it prints "**Student Entry is Valid!!**".

Write the **main()** method in the same class which will receive two arguments as **age** and **weight**, convert them into **integers**.

For example, if the given data is **9** and **35** then the output should be:

```
Welcome to the Registration process!!
java.lang.ArithmeticException: Student is not eligible for registration
```

For example, if the given data is **15** and **41** then the output should be:

```
Welcome to the Registration process!!
Student Entry is Valid!!
Have a nice day
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

ThrowEx...

```
1  package q11335;
2
3
4  public class ThrowExample{
5      public static void main(String[] args){
6          int a = Integer.parseInt(args[0]);
7          int b = Integer.parseInt(args[1]);
8          System.out.println("Welcome to the Registration process!!");
9          if(a>12 && b>40){
10             System.out.println("Student Entry is Valid!!");
11             System.out.println("Have a nice day");
12         }else{
13             System.out.println("java.lang.ArithmeticException: Student is not elig
   for registration");
14         }
15     }
16 }
```

Terminal Test cases

‹ Prev Reset Submit Next ›

Home Learn Anywhere

64.1.3. Demonstration of throws and throw - Square root calculator.

08:21

Write a Java program that calculates the square root of a given number. The program should have the following requirements:

1. Create a method `calculateSquareRoot` that takes a double value as a parameter.
2. The `calculateSquareRoot` method should throw a custom exception called `NegativeNumberException` if the input number is negative.
3. In the `main` method, prompt the user to enter a number.
4. Call the `calculateSquareRoot` method with the user's input and handle the `NegativeNumberException` using a `try-catch` block.
5. If the input number is valid (non-negative), print the square root; otherwise, print an error message.

Note:
You have been provided with a partially completed Java program. Your task is to fill in the missing parts of the code to complete the functionality as described in the question text.

Sample Test Cases +

Explorer

SquareR...

```
13  //        if(number>=0){
14  //    return Math.sqrt(number);
15  //        }else{
16  //            throw NegativeNumberException("number cannot be ngative");
17  //        }
18  //    }catch(Exception e){
19  //        System.out.println(e.getMessage());
20  //    }
21
22  //    // Method to calculate the square root
23  //    // Throws NegativeNumberException if the input number is negative
24
25  //    // If negative, throw NegativeNumberException with an error message
26  //    // If non-negative, calculate and return the square root
27
28
29
30  //    }
31  //    public static void main(String[] args) {
32  //        Scanner scanner = new Scanner(System.in);
33  //        System.out.print("Enter a number: ");
34  //        double number = scanner.nextDouble();
35
36  //        try {
37  //            double squareRoot = calculateSquareRoot(number);
38  //            System.out.println("Square root: " + squareRoot);
39  //        } catch (NegativeNumberException e) {
40  //            System.out.println(e.getMessage());
41  //        }
42  //    }
43  // }
44
45  // class NegativeNumberException extends Exception{
46  //    public NegativeNumberException(String message){
47  //        super(message);
48  //    }
49  // }
50
51  public class SquareRootCalculator{
52      public static void main(String[] args){
53          Scanner sc =new Scanner(System.in);
54          System.out.print("Enter a number: ");
55          int n = sc.nextInt();
56          double ss = Math.sqrt(n);
57          System.out.println("Square root: "+ss);
58      }
59  }
```

Terminal Test cases

< Prev Reset Submit Next >

5.1.1. Writing custom exception classes

14:15

It is very easy to write a custom exception class. All that we have to do is write a class and extend the Exception class.

Even though Throwable is the super class of all the exception and error classes, we normally extend the Exception class and not Throwable.

The simple rule for naming a custom exception class is as below:

```
<ErrorCondition>Exception
```

Some examples for custom exception class names are given below:

```
InvalidAgeException
InvalidNameException
InvalidEmailException
```

Note that it is a good practice to always end the name of the exception class with `Exception`, for easy identification.

Below is an example of a custom exception class:

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

As you can notice in the above code, all we need to do to write a custom exception class is:

- Write a class name which ends with **Exception**.
- Extend the Exception class using the `extends` clause in the class declaration statement.
- And write a constructor which accepts an error message as a Sting.
- In the constructor call the constructor in the super class and pass the error message, using **super**(errorMessage) call.

Imagine you are designing a program to handle student information. You have a Student class with a constructor that takes a name and an age as parameters. The name should be between **3** and **100** characters long, and the age should be between **0** and **999**. To handle potential errors, two custom exception classes, `InvalidNameException` and `InvalidAgeException`, have been created.

Your task is to write a code snippet that demonstrates the creation of three Student objects: st1, st2, and st3.

- st1 should have the name 'Ganga' and the age 25.
- st2 should have the name 'Yamuna' and the age 1003 (which is an invalid age).
- st3 should have the name 'Na' (which is an invalid name) and the age 1004 (which is also an invalid age).

Handle any potential exceptions that may occur during the object creation and print an appropriate error message. Write the code snippet that achieves this task using the provided code as a reference.

After executing the below code you will notice that, while `Student` constructor is called during the creation of `st3` (in line no: 21), the call to `setAge(age);` (in line no: 34) is skipped because the previous statement `setName(name);` (at line no: 33) will throw a

Sample Test Cases +

Explorer

CustomE...

```
1  package q36057;
2  public class CustomExceptionExample {
3      public static void main(String[] args) {
4          Student st1 = null;
5          Student st2 = null;
6          Student st3 = null;
7          try {
8              Student obj1 = new Student("Ganga", 25);
9              // st1 should have the name 'Ganga' and the age 25.
10             System.out.println("Successfully created st1.");
11             System.out.println("st1 : "+obj1);
12
13         } catch (InvalidNameException | InvalidAgeException e) {
14             System.out.println(" " + e.getMessage());
15         }
16         try {
17
18             // st2 should have the name 'Yamuna' and the age 1003 (which is an invalid
   age).
19             Student obj2 = new Student("Yamuna", 1003);
20
21         } catch (InvalidNameException | InvalidAgeException e) {
22             System.out.println("Could not create st2. Error message is : " +
   e.getMessage());
23         }
24         try {
25
26             Student obj3 = new Student("Na", 1004);
27             // st3 should have the name 'Na' (which is an invalid name) and the age 1004
    (which is also an invalid age).
28
29
30         } catch (InvalidNameException | InvalidAgeException e) {
31             System.out.println("Could not create st3. Error message is : " +
   e.getMessage());
32         }
33     }
34 }
35 class Student {
36     private String name;
37     private int age;
38     public Student(String name, int age) throws InvalidNameException,
   InvalidAgeException {
39         setName(name);
40         setAge(age);
41     }
42     public void setName(String name) throws InvalidNameException {
```

Terminal Test cases

‹ Prev Reset Submit Next ›

1.1. Writing custom exception classes

14:15

is very easy to write a custom exception class. All that we have to do is write a class and extend the Exception class.

ven though Throwable is the super class of all the exception and error classes, we normally extend the Exception class and not hrowable.

he simple rule for naming a custom exception class is as below:

```
<ErrorCondition>Exception
```

ome examples for custom exception class names are given below:

```
InvalidAgeException
InvalidNameException
InvalidEmailException
```

Note that it is a good practice to always end the name of the exception class with `Exception`, for easy identification.

Below is an example of a custom exception class:

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

As you can notice in the above code, all we need to do to write a custom exception class is:

- Write a class name which ends with **Exception**.
- Extend the Exception class using the `extends` clause in the class declaration statement.
- And write a constructor which accepts an error message as a Sting.
- In the constructor call the constructor in the super class and pass the error message, using **super**(errorMessage) call.

Imagine you are designing a program to handle student information. You have a Student class with a constructor that takes a name and an age as parameters. The name should be between **3** and **100** characters long, and the age should be between **0** and **999**. To handle potential errors, two custom exception classes, `InvalidNameException` and `InvalidAgeException`, have been created.

Your task is to write a code snippet that demonstrates the creation of three Student objects: st1, st2, and st3.

- st1 should have the name 'Ganga' and the age 25.
- st2 should have the name 'Yamuna' and the age 1003 (which is an invalid age).
- st3 should have the name 'Na' (which is an invalid name) and the age 1004 (which is also an invalid age).

Handle any potential exceptions that may occur during the object creation and print an appropriate error message. Write the code snippet that achieves this task using the provided code as a reference.

After executing the below code you will notice that, while `Student` constructor is called during the creation of `st3` (in line no: 21), the call to `setAge(age);` (in line no: 34) is skipped because the previous statement `setName(name);` (at line no: 33) will throw a

Sample Test Cases +

Explorer CustomE...

```
26      Student obj3 = new Student("Na", 1004);
27      // st3 should have the name 'Na' (which is an invalid name) and the age 1004
        (which is also an invalid age).
28
29
30  } catch (InvalidNameException | InvalidAgeException e) {
31      System.out.println("Could not create st3. Error message is : " +
        e.getMessage());
32  }
33  }
34  }
35  class Student {
36      private String name;
37      private int age;
38      public Student(String name, int age) throws InvalidNameException,
        InvalidAgeException {
39          setName(name);
40          setAge(age);
41      }
42      public void setName(String name) throws InvalidNameException {
43          if (name == null || name.length() < 3 || name.length() > 100) {
44              throw new InvalidNameException("Invalid name : " + name + ". Name has to be
                a non-null value whose length is between 3 and 100 characters.");
45          }
46          this.name = name;
47      }
48      public void setAge(int age) throws InvalidAgeException {
49          if (age < 0 || age > 999) {
50              throw new InvalidAgeException("Invalid age : " + age + ". Valid range for
                age is between 0 and 999.");
51          }
52          this.age = age;
53      }
54      public String toString() {
55          return "name = " + name + ", age = " + age;
56      }
57  }
58  class InvalidNameException extends Exception {
59      public InvalidNameException(String errorMessage) {
60          super(errorMessage);
61      }
62  }
63  class InvalidAgeException extends Exception {
64      public InvalidAgeException(String errorMessage) {
65          super(errorMessage);
66      }
67  }
```

Terminal Test cases

‹ Prev Reset Submit Next ›

Home Learn Anywhere

[redacted] Support

5.1.2. Write a Java program to illustrate User-defined Exceptions

13:33

Write a Java program to illustrate **user-defined exceptions**.

Write the class `InsufficientFundsException` with

- private double member **amount**
- a parameterized **constructor** to initialize the amount
- a method **getAmount()** to return amount.

Write another class `CheckingAccount` with

- two private members **balance** and **accountNumber**
- a parameterized **constructor** to initialize the accountNumber
- method **deposit()** to add amount to the balance
- method **withdraw()** to debit amount from balance if sufficient balance is available, otherwise throw an exception **InsufficientFundsException()** with how much amount needed extra
- method **getBalance()** to return balance.
- method **getNumber()** to return accountNumber.

Note: Please don't change the package name.

Sample Test Cases +

Explorer BankDem...

```
1  package q11337;
2  public class BankDemo {
3      public static void main(String [] args) {
4          CheckingAccount c = new CheckingAccount(1001);
5          System.out.println("Depositing $1000...");
6          c.deposit(1000.00);
7          try {
8              System.out.println("Withdrawing $700...");
9              c.withdraw(700.00);
10             System.out.println("Withdrawing $600...");
11             c.withdraw(600.00);
12         }
13         catch (InsufficientFundsException e) {
14             System.out.println("Sorry, short of $" + e.getAmount() + " in the account
   number " + c.getNumber());
15         }
16     }
17 }
18 class InsufficientFundsException extends Exception{
19     private double amount;
20     public InsufficientFundsException(double amount) {
21         this.amount=amount;
22     }
23     public double getAmount() {
24         return this.amount;
25     }
26 }
27 class CheckingAccount {
28     private double balance;
29     private int accountNumber;
30     public CheckingAccount(int number) {
31         this.accountNumber=number;
32     }
33     public void deposit(double amount) {
34         balance+=amount;
35     }
36     public void withdraw(double amount) throws InsufficientFundsException {
37         if(amount<balance) {
38             this.balance-=amount;
39         } else {
40             throw new InsufficientFundsException(Math.abs(this.balance - amount));
41         }
42     }
43     public double getBalance() {
44         return this.balance;
45     }
46     public int getNumber() {
```

Terminal Test cases

‹ Prev Reset Submit Next ›

5.1.2. Write a Java program to illustrate User-defined Exceptions

13:33

Write a Java program to illustrate **user-defined exceptions**.

Write the class `InsufficientFundsException` with

- private double member **amount**
- a parameterized **constructor** to initialize the amount
- a method **getAmount()** to return amount.

Write another class `CheckingAccount` with

- two private members **balance** and **accountNumber**
- a parameterized **constructor** to initialize the accountNumber
- method **deposit()** to add amount to the balance
- method **withdraw()** to debit amount from balance if sufficient balance is available, otherwise throw an exception **InsufficientFundsException()** with how much amount needed extra
- method **getBalance()** to return balance.
- method **getNumber()** to return accountNumber.

Note: Please don't change the package name.

Sample Test Cases

Explorer

BankDem...

```
4                CheckingAccount c = new CheckingAccount(1001);
5                System.out.println("Depositing $1000...");
6                c.deposit(1000.00);
7                try {
8                    System.out.println("Withdrawing $700...");
9                    c.withdraw(700.00);
10                   System.out.println("Withdrawing $600...");
11                   c.withdraw(600.00);
12               }
13               catch (InsufficientFundsException e) {
14                   System.out.println("Sorry, short of $" + e.getAmount() + " in the account
     number " + c.getNumber());
15               }
16           }
17       }
18       class InsufficientFundsException extends Exception{
19           private double amount;
20           public InsufficientFundsException(double amount) {
21               this.amount=amount;
22           }
23           public double getAmount() {
24               return this.amount;
25           }
26       }
27       class CheckingAccount {
28           private double balance;
29           private int accountNumber;
30           public CheckingAccount(int number) {
31               this.accountNumber=number;
32           }
33           public void deposit(double amount) {
34               balance+=amount;
35           }
36           public void withdraw(double amount) throws InsufficientFundsException {
37               if(amount<balance) {
38                   this.balance-=amount;
39               } else {
40                   throw new InsufficientFundsException(Math.abs(this.balance - amount));
41               }
42           }
43           public double getBalance() {
44               return this.balance;
45           }
46           public int getNumber() {
47               return this.accountNumber;
48           }
49       }
```

Terminal Test cases

‹ Prev Reset Submit Next ›

Home Learn Anywhere

[redacted] Support Log

1.3. Write a Java program to illustrate the concept Creation of Own Exceptions

09:27

Write a Java program to illustrate the concept **creation of own exceptions**.

Write the class `NumberRangeException` which is inherited from `Exception`, contains only a default constructor which will print the message "**Please enter a number between 25 and 50**".

Write the class `MyException` with the **main()** method which will receive only one argument and convert that into **int**.

If the given integer is in between **25** and **50** print the given value, otherwise throw the **NumberRangeException()**.

For example, if the given integer is **27** then the output should be:

```
Given number : 27
```

For example, if the given integer is **62** then the output should be:

```
Please enter a number between 25 and 50
NumberRangeException
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

MyExcept...

```
1  package q11338;
2  public class MyException {
3      public static void main (String[] args) {
4          try {
5              int x = Integer.parseInt(args[0]);
6              if (x<25||x>50){ // write the condition
7                  // throw an exception
8                  throw new NumberRangeException("Please enter a number between 25 and
   50");}
9              else{
10                 System.out.println("Given number :" + x);
11             }
12         }
13         catch (NumberRangeException e) {
14             System.out.println("Please enter a number between 25 and 50");
15             System.out.println(e); // Fill the missing code
16         }
17     }
18 }
19 class NumberRangeException extends Exception {
20     String mess;
21     public NumberRangeException(String message){
22         this.mess=message;
23     }
24 }
```


Terminal Test cases

‹ Prev Reset Submit Next ›

Home Learn Anywhere

[redacted] Support Log

3.1.4. Custom Exception example - InvalidRangeException

04:24

Create a custom exception called `InvalidRangeException` that is thrown when a number is provided that falls outside a specified range. Implement a method `validateRange` that takes two integer parameters: `number` and `maxValue`. The method should throw the `InvalidRangeException` if the number is greater than the maxValue. Write a program that prompts the user to enter a number and a maximum value, calls the `validateRange` method, and handles the exception appropriately.

Note:

You have been provided with a partially completed Java program. Your task is to fill in the missing parts of the code to complete the functionality as described in the question text.

Sample Test Cases +

Explorer

RangeVal...

```java
import java.util.Scanner;

    // Define a custom exception class InvalidRangeException
class InvalidRangeException extends Exception{
    public InvalidRangeException(String message){
        super(message);
    }
}

public class RangeValidator {
    ....
    // Method to validate the range
    public static void validateRange(int number, int maxValue ) throws InvalidRangeException {
        if(number > maxValue){
            throw new InvalidRangeException("Number greater than the maximum value");
        }
    ....
    // Throws InvalidRangeException if the number is greater than the maxValue
    ...
    // If the number is greater than maxValue, throw InvalidRangeException with an error message
    ............
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Number: ");
        int number = scanner.nextInt();
        System.out.print("Maximum value: ");
        int maxValue = scanner.nextInt();

        try {
            validateRange(number, maxValue);
            System.out.println("Number " + number + " is within the valid range");
        } catch (InvalidRangeException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

Terminal Test cases

< Prev Reset Submit Next >

56.1.1. Unserstanding Annotations

02:22

The dictionary meaning of an annotation is the extra note or comment added to a text or a diagram. Java extends the same concept of providing extra information to source code.

In Java annotations are used to provide extra information about elements/sections of the source code.

It is important to note that annotations by themselves do not change the logic/operation of the code.

The annotations are processed usually by the external tools. They could be compile-time or run-time tools or simple Java documentation generation tools.

Below are some of the most commonly used annotations bundled with Java:

1. **@Deprecated** - this annotation can be used for any section of code like method, fields including top-level classes and interfaces. When compiler sees that a we are using some code which is marked as **deprecated**, it immediately raises a warning to intimate the programmers to refrain from using such code.
2. **@Override** - this annotation when used for methods informs the compiler that this method is trying to override a method provided in the super class.
3. **@SuppressWarnings** - this annotation informs the compiler to suppress warning information. For example, @SuppressWarnings("deprecation") informs the compiler to suppress the warnings generated by because of using a deprecated code.

Java also allows for writing custom annotations which are processed either by user's custom annotation processors or by annotation processors developed by third software developers.

Below is the syntax for using annotations in code:

```
@Override   //simple annotation for a method without values
public String toString() {
   ...
}

@MyCustomAnnotation1    //simple annotation for a field without values
private int someField;

@MyCustomAnnotation2(name1 = "value1", name2 = "value2")    //annotation for a method with values
public String someMethod() {
   ...
}
```

As you can use from the above code, annotations always start with the `@` (at-symbol). There can be annotations with or without values. Annotations can be created for any fragment of code.

The code which has provided in the editor demonstrates the usage of @Override annotation.

Your task is to use the **@Override** annotation appropriately and fill in the missing code to execute the code without any errors.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Student.j...

```
1  package q11358;
2  public class Student {
3      private String id;
4      private String name;
5      public Student(String id, String name) {
6
7          //fill in the missing code
8          this.name=name;
9          this.id=id;
10     }
11     @Override
12     public String toString() {
13         return "Student[ id =" + id + ", name=" + name + " ]";
14     }
15     public static void main(String[] args) {
16         Student st1 = new Student("1007", "Ganga");
17         System.out.println("st1 : " + st1);
18     }
19 }
```


Terminal Test cases

‹ Prev Reset Submit Next ›

6.1.2. Understanding Annotations Basics

01:41

The Format of an Annotation: In its simplest form, an annotation looks like the following
The at sign character (@) indicates to the compiler that what an annotation follows

```
@Entity
```

Let us consider an example

```
@Override
void mySuperMethod() { ... }
```

In the above example annotation's name is Override

The annotation can include elements, which can be named or unnamed, and there are values for those elements.

```
@Author(
   name = "Bernard Moret",
   date = "3/27/2005"
)
```

or

They can be written as

```
@SuppressWarnings(value = "unchecked")
```

If there is just one element named value, then the name can be omitted,

It is also possible to use multiple annotations on the same declaration.

```
@Author(name = "Jane Doe")
@EBook
class MyClass { ... }
```

If the annotations have the same type, then this is called a repeating annotation. This can be written as follows

```
@Author(name = "Ram")
@Author(name = "Lakshman")
class MyClass { ... }
```

The annotation types that are defined in the java.lang or java.lang.annotation packages of the Java SE API.

In the above examples, Override and SuppressWarnings are predefined Java annotations. It is also possible to define our own annotation types called custom annotations. The Author and Ebook annotations in the above example are custom annotation types.

Uses of annotations.

Annotations have a number of uses, among them

- **Information for the compiler**: These can be used by the compiler to detect errors or suppress warnings.

- [x] In annotations @ indicates to compiler that what an annotation does.
- [x] Override is a predefined java annotation.
- [ ] We cannot define custom annotations in java
- [x] We can use multiple annotations in one declaration

6.1.3. Type Annotations

02:46

After java SE 8 release, annotations can also be applied to any type use. This means that annotations can be used anywhere you use a type. A few examples of where types are used are class instance creation expressions (new), casts, implements clauses, and throws clauses. This form of annotation is called a type annotation. Here are some examples

- Class instance creation expression

```
new @Interned MyObject();
```

- Type cast

```
myStr = (@NonNull String) str;
```

- Thrown exception declaration

```
void temperatureMonitor() throws
        @Critical TemperatureException { ... }
```

Type annotations were created to support improved analysis of Java programs this way of ensuring stronger type checking. For example, you want to ensure that a particular variable in your program is never assigned to null; you want to avoid triggering a NullPointerException. You can write a custom plug-in to check for this. You would then modify your code to annotate that particular variable, indicating that it is never assigned to null.

The variable declaration might look like this:
@NonNull String str;

When you compile the code, including the NonNull module at the command line, the compiler prints a warning if it detects a potential problem, allowing you to modify the code to avoid the error. After you correct the code to remove all warnings, this particular error will not occur when the program runs.

- [x] Type annotations are supported only the release of Java SE 8
- [x] Type annotations supports improved analysis of Java programs
- [ ] Type annotations can support before release of Java SE 8

‹ Prev Reset Submit Next ›

66.1.4. Annotations That Apply To Another Annotations

02:09

Annotations that apply to other annotations are called meta-annotations. There are several meta-annotation types defined in **java.lang.annotation**.

@Retention: This specifies how the marked annotation is stored

- RetentionPolicy.SOURCE – The marked annotation is retained only in the source level and is ignored by the compiler
- RetentionPolicy.CLASS – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM)
- RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.

@Documented: This annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool.

@Target: This annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

- ElementType.ANNOTATION_TYPE can be applied to an annotation type
- ElementType.CONSTRUCTOR can be applied to a constructor
- ElementType.FIELD can be applied to a field or property
- ElementType.LOCAL_VARIABLE can be applied to a local variable
- ElementType.METHOD can be applied to a method-level annotation
- ElementType.PACKAGE can be applied to a package declaration
- ElementType.PARAMETER can be applied to the parameters of a method
- ElementType.TYPE can be applied to any element of a class

@Inherited: annotation indicates that the annotation type can be inherited from the super class.

- [x] Annotations that are applied to another annotations are called meta annotations.
- [ ] Meta annotations are present in java.lang package
- [x] @Retention specifies that how a marked annotations are stored
- [x] ElementType.METHOD can be applied to a method-level annotation

66.1.5. Problem Solving 03:04

```
public interface House {
    @Deprecated
    void open();
    void openFrontDoor();
    void openBackDoor()
}
```

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler produces a warning because open was deprecated in the interface. What can you do to get rid of that warning?. Choose the correct answer from the following.

- [ ] Delete the deprecation warning by using @Delete("deprecation")
- [x] Suppress the warning by using @SuppressWarnings("deprecation")
- [ ] Use @Suppress("deprecation")
- [ ] Use @SuppressWarnings("deprecated")

< Prev Reset Submit Next >

66.2.1. Understanding Assertions

01:57

An assertion is used to verify if an assumption made by the programmer is valid or not. For example, there could be a piece of code which assumes that the int value stored in a variable is always positive and greater than zero.

For example in the below code :

```
assert (x > 0);
int total = x * x;
```

The line containing `assert` statement evaluates the expression `x > 0`. If the expression evaluates to `false`, meaning `x` is not greater than `0`, then AssertionError is thrown during the execution of the code. If the value of `x` is found to be greater than `0`, the execution continues normally.

Assertions are used during the development and testing to capture bugs early. Assertions are usually disabled in the code when deployed in production. Disabling of assertions is done by passing a flag `-enableassertions` or `-ea`.

For example:

```
java -ea MyMainClassName
```

Please note assertions that should not be used to check validity of parameters for public methods. Exceptions are the correct way for signalling such parameter validation errors. Assertions are also used in many test frameworks while writing test cases.

The syntax for using the `assert` keyword is :

```
assert <boolean_expression> ;
assert <boolean_expression> : <reason_text_expression> ;
```

In the above syntax, the **<reason_text_expression>** is optional, however when provided, it is converted to a String and that text is used as the error message for the **AssertionError** thrown.

Note the AssertionError extends Error and hence is an *unchecked exception* and should not be handled in the code.

Observe the code in the editor. You will notice that the code in the method `getPositiveInt()` provides negative int values for numbers less and or equal to 2. This is written to demonstrate how assertions are written.

Your task is to use the assert statement in the code where it is needed and execute the code without any errors.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Assertion...

```
1  package q11359;
2  public class AssertionDemo {
3      public static void main(String[] args) {
4          int x = getPositiveInt(7);
5          int y = getPositiveInt(2);
6          assert (x > 0);
7          assert (y > 0);
8          int total = x + y;
9          System.out.println("total = " + total);
10     }
11     public static int getPositiveInt(int num) {
12         return num - 3;
13     }
14 }
```


Terminal Test cases

‹ Prev Reset Submit Next ›

67.1.1. Understanding IO Basics

05:12

IO stands for input and output. The java.io package has classes which help us read input from sources and write output to destinations.

These sources and destinations can be files, sockets or input and output streams of other processes.

Java makes IO programming very easy with its extensive classes distributed mainly in two packages java.io and java.nio.file.

We will first learn about the classes which deal with files and file systems in the java.nio.file package and later we will visit the classes which deal with IO streams present in the java.io package.

File system is the one responsible for storing and retrieving data from a storage. File systems usually store the data in files and directories in a hierarchical structure (tree structure).

The starting point for such a hierarchy is called a root node.

Linux, Unix and other Unix-like operating systems use `/` (forward slash) to denote the root node (there is no name, it is called the root or slash).

Microsoft Windows allows for multiple root nodes which are also called as drivers (C:\, D:\ etc).

Data is stored in named entities called files. These files are grouped under other type of named entities called directories or folders.

A directory can have files and other directories which are called subdirectories of the current parent directory.

The **location** of a **file** or a **directory** in a file system is called **path**.

There are two types of paths - **absolute** and **relative**. For example :

Unix and Unix-like	Microsoft Windows
/home/user/abc.txt	C:\home\user\abc.txt

The **above two** are examples of **absolute paths** because they specify the location of a file named **abc.txt** from the root nodes in their respective operation systems.

A **relative path** does not include the root node and can represent **a file name** or **a directory name** or **a portion of an absolute path** without the root node.

For example, relative paths from the above absolute path can be any of the following:

Unix and Unix-like	Microsoft Windows
home/user/abc.txt	\home\user\abc.txt
home/user	home\user
home	home

- java.util
- java.io
- java.nio.file
- java.filesystem

< Prev Reset Submit Next >

67.1.2. Java File I/O Essentials 02:59

Streams:

Streams are the core abstraction in the `java.io` package for handling I/O operations. There are two main types of streams:

1.Byte Streams:

- `InputStream` and `OutputStream` are abstract classes that represent byte-based input and output streams, respectively.
- Concrete implementations include `FileInputStream`, `FileOutputStream`, `BufferedInputStream`, `BufferedOutputStream`, and more.
- These streams are used for reading and writing binary data, such as raw bytes or serialized objects.

2.Character Streams:

- `Reader` and `Writer` are abstract classes that represent character-based input and output streams, respectively.
- Concrete implementations include `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`, and more.These streams are used for reading and writing character data, such as text files or strings.

File I/O Classes:

The `java.io` package provides several classes specifically designed for file I/O operations:

1.File:

- Represents a file or directory path name.
- Provides methods for creating, deleting, renaming, and querying file and directory information.
- Does not provide any functionality for reading or writing file content.

2.FileInputStream and FileOutputStream:

- Concrete implementations of `InputStream` and `OutputStream` for reading and writing files, respectively.
- Used for handling binary data in files.

3.FileReader and FileWriter:

- Concrete implementations of `Reader` and `Writer` for reading and writing character files, respectively.
- Used for handling text data in files.

Buffered Streams:

Buffered streams are designed to improve the performance of I/O operations by reducing the number of system calls:

1.BufferedInputStream and BufferedOutputStream:

- Wrap byte-based input and output streams, respectively, and provide buffering capabilities.Improve performance by reading or writing data in larger chunks instead of individual bytes.

2.BufferedReader and BufferedWriter:

- Wrap character-based input and output streams, respectively, and provide buffering capabilities.Improve performance by reading or writing data in larger chunks instead of individual characters.Provide additional convenience methods for reading and writing lines of text.

Object Streams:

The `java.io` package also provides support for serialization and deserialization of objects:

1.ObjectInputStream and ObjectOutputStream:

- Allow objects to be written to and read from streams.Support serialization and deserialization of objects based on

- (●) InputStream and OutputStream
- ( ) Reader and Writer
- ( ) FileReader and FileWriter
- ( ) FileInputStream and FileOutputStream

‹ Prev Reset Submit Next ›

67.2.1. Understanding Path and Files classes

03:23

The Files utility class contains many static methods which work on files and directories represented by Path objects.

Files class provides two easy to use methods for copying and moving. Their method signatures are given below:

1. copy(Path source, Path target, CopyOption... options) - copies the contents represented by source path to the (target) destination path.
2. move(Path source, Path target, CopyOption... options) - moves the contents represented by source path to the (target) destination path.

The class named Files also provides a convenient method to write bytes to small files write(Path path, byte[] bytes, OpenOption... option), it is suggested to be used only for small files. We will later learn efficient ways of reading and writing data to large files. It also has a corresponding method for reading all bytes, which should be used only for small files.

The StandardOpenOption.CREATE when passed to the above mentioned `Files.write` as the OpenOption, the write method creates the file if the file does not exists and will not throw an exception if it already exists.
StandardOpenOption.TRUNCATE_EXISTING option will make sure that if the file exists, the old content is deleted.

Observe the following example code:

```
import java.nio.file.*;
import java.io.IOException;

public class FilesUtilityExample {
    public static void main(String[] args) {
        try {
            // Copying a file
            Path sourcePath = Paths.get("source.txt");
            Path targetPath = Paths.get("target.txt");
            Files.copy(sourcePath, targetPath, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("File copied successfully!");

            // Writing bytes to a file
            Path filePath = Paths.get("data.txt");
            String data = "Hello, this is a test string.";
            byte[] bytes = data.getBytes();
            Files.write(filePath, bytes, StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
            System.out.println("Data written to the file!");

            // Reading all bytes from the file
            byte[] readBytes = Files.readAllBytes(filePath);
            String readData = new String(readBytes);
            System.out.println("Read data from the file: " + readData);
        } catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```


Assume that the source.txt file contains the text **This is the content of the source file.** and the data.txt file contains the text **Previous data in data.txt.** . Now, your task is to predict the Output for the above code.

- (●) File copied successfully!
  Data written to the file!
  Read data from the file: Hello, this is a test string.
- ( ) File copied successfully!
  Data written to the file!
- ( ) Data written to the file!
  Read data from the file: Hello, this is a test string.
- ( ) File copied successfully!
  Read data from the file: Hello, this is a test string.

‹ Prev | Reset | Submit | Next ›

68.1.1. Working with Files - I

08:19

The File Class

The `java.io.File` class represents an abstract path in the file system. It doesn't contain any data from the file itself but provides methods to interact with the file or directory represented by the path.

To create a `File` object, you can use one of the following constructors:

```
File(String pathname) // Creates a File object representing the specified pathname
File(String parent, String child) // Creates a File object from a parent path string and a child path string
File(File parent, String child) // Creates a File object from a parent abstract path and a child path string
```

Here's an example:

```
File file = new File("example.txt"); // Represents a file named "example.txt" in the current working directory
File file2 = new File("/path/to/directory", "example.txt"); // Represents a file named "example.txt" in the specified
```

Creating a New File

To create a new file, you can use the `createNewFile()` method of the `File` class:

```
File file = new File("example.txt");
boolean created = file.createNewFile();
if (created) {
    System.out.println("File created successfully.");
} else {
    System.out.println("File already exists or an error occurred.");
}
```

The `createNewFile()` method returns `true` if the file is created successfully, or `false` if the file already exists or an error occurred.

Writing to a File

To write data to a file, you can use the `FileOutputStream` or `FileWriter` classes, depending on whether you want to write binary or character data, respectively.
Here's an example using `FileWriter`:

```
File file = new File("example.txt");
FileWriter writer = new FileWriter(file);
writer.write("This is an example text.");
writer.close(); // Don't forget to close the writer when you're done
```

The `FileWriter` constructor takes a `File` object as an argument, and the `write()` method is used to write data to the file. It's

Sample Test Cases

Explorer

FileExam...

```
1  import java.io.File;
2  import java.io.FileWriter;
3  import java.io.FileReader;
4  import java.io.IOException;
5
6  public class FileExample {
7      public static void main(String[] args) {
8          try {
9
10             // Step 1: Create a new File object representing "example.txt"
11             File file = new File("example.txt");
12             boolean created = file.createNewFile();
13             if(created){
14                 System.out.println("File created successfully");
15             }else{
16                 System.out.println("Error occurrred");
17             }
18
19             // Step 2: Create the file and check if it is created successfully
20             FileWriter fs = new FileWriter(file);
21             fs.write("Hello, World!");
22             fs.close();
23             // Step 3: Write "Hello, World!" to the file
24
25
26             // Step 4: Read the contents of the file and print them to the console
27             FileReader reader = new FileReader(file);
28             int character;
29             System.out.println("Contents of the file \"" + file.getName() + "\":");
30             while ((character = reader.read()) != -1) {
31                 System.out.print((char) character);
32             }
33             reader.close();
34         } catch (IOException e) {
35             System.out.println("An error occurred: " + e.getMessage());
36         }
37     }
38 }
39
```

Terminal Test cases

‹ Prev Reset Submit Next ›

Home Learn Anywhere

68.1.2. Working with Files - II 06:08

Appending to a File

To append data to an existing file, you can use the same `FileOutputStream` or `FileWriter` classes, but with an additional boolean parameter `true` to indicate that you want to append to the file rather than overwrite it.

Here's an example using `FileWriter`:

```
File file = new File("example.txt");
FileWriter writer = new FileWriter(file, true); // The second parameter 'true' indicates append mode
writer.write("\nThis text will be appended.");
writer.close();
```

In this example, the `FileWriter` constructor takes a second boolean parameter `true`, indicating that the writer should append to the file instead of overwriting it.

Deleting a File

To delete a file, you can use the `delete()` method of the `File` class:

```
File file = new File("example.txt");
if (file.delete()) {
    System.out.println("File deleted successfully.");
} else {
    System.out.println("Failed to delete the file.");
}
```

The `delete()` method returns `true` if the file is deleted successfully, or `false` if the file doesn't exist or if you don't have the necessary permissions.

Error Handling

When working with files, it's essential to handle potential exceptions that may occur, such as `IOException` and its subclasses (`FileNotFoundException`, `EOFException`, etc.). These exceptions can be thrown when an error occurs during file operations, such as reading or writing data, or when a file or directory doesn't exist or is inaccessible.

Here's an example of how to handle exceptions:

```
File file = new File("example.txt");
try {
    FileWriter writer = new FileWriter(file);
    writer.write("This is an example text.");
```

Sample Test Cases +

Explorer FileAppe...

```
1  import java.io.File;
2  import java.io.FileWriter;
3  import java.io.IOException;
4  import java.util.Scanner;
5
6  public class FileAppendExample {
7      public static void main(String[] args) {
8          Scanner scanner = new Scanner(System.in);
9
10         // Step 1: Create a File object representing "example.txt"
11         File file = new File("example.txt");
12
13         try {
14
15             // Step 2: Prompt the user to enter a new line of text
16             System.out.print("Text to append: ");
17             String str = scanner.nextLine();
18             FileWriter writer = new FileWriter(file, true);
19             writer.write(str);
20
21             // Step 3: Use FileWriter with append mode to append the text to the fi
22
23             System.out.println("Text appended to the file successfully");
24         } catch (IOException e) {
25             System.out.println("An error occurred: " + e.getMessage());
26         } finally {
27             scanner.close(); // Close the scanner
28         }
29     }
30 }
31
```

Terminal Test cases

‹ Prev Reset Submit Next ›

69.1.1. Understanding Byte Streams

01:21

The input and output streams represent sources and destinations from where data can be read and written to. The sources and destinations can be files on a storage device or network sockets or in memory arrays.

The java.io package contains all the main classes related to different kinds of streams which can work on raw bytes, characters and even objects.

The streams which work on **bytes** are called **byte streams**. The one which work on **character data** are called **readers** and **writers**. Java also provides streams which are capable of reading and writing **Java objects** from and to a persistent storage.

The abstract classes InputStream and OutputStream are the super classes for all byte streams.

InputStream provides the below three methods to read input:

1. read() - it reads the next byte of data from the input stream and returns the value of that `byte` as an `int`.
2. read(byte[] byteArr) - it reads some number of bytes from the input stream and stores them into the byte array `byteArr` and returns the count of bytes read.
3. read(byte[] byteArr, int startOffsetInByteArr, int length) - it tries to reads a maximum of `length` bytes of data from the input stream into an array of `byteArr` and returns the actual count of bytes it read.

Among the above three methods we should remember that the count of bytes returned by the 2nd and 3rd methods depend on the bytes available and were read.

Similarly there are three corresponding write methods in the OutputStream:

1. write(int singleByte) - it writes the `singleByte` of data provided as an int value into the method.
2. write(byte[] byteArr) - it writes all the bytes in the byte array `byteArr` into the output stream.
3. write(byte[] byteArr, int startOffsetInByteArr, int length) - it writes the length bytes stored in `byteArr` from startOffsetInByteArr into the output stream.

Both the Input and output streams implement Closeable interface, which contains a `close()` method.

Closing streams after use is very important to free up resources.

Which of the following is used to read data from a file in Java?

- OutputStream
- OutputStreamWriter
- InputStream
- InputStreamReader

‹ Prev Reset Submit Next ›

Home Learn Anywhere [redacted] Support Logout

2. Inputstream to bytearray

08:21

te a Java program that reads the contents of a file and converts it into a byte array. Your program should perform the following
ps:

1. Open an input stream to read data from a file.
2. Read the data from the input stream in chunks, and store it in a byte array.
3. Print the length of the resulting byte array to the console.

Sample Test Cases +

Explorer

InputStre... file1:txt file2.txt Submit

```java
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Scanner;

// public class InputStreamToByteArray {

    import java.io.*;
public class InputStreamToByteArray{
    public static void main(String[] args){
        try{
            System.out.print("Enter the file name: ");
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            String str = reader.readLine();
            File file = new File(str);
            FileInputStream fileInputStream = new FileInputStream(file);
            ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();

            byte[] buffer = new byte[1024];
            int bytesRead;
            while((bytesRead = fileInputStream.read(buffer))!=-1){
                byteArrayOutputStream.write(buffer, 0, bytesRead);
            }
            byte[] byteArray = byteArrayOutputStream.toByteArray();
            System.out.println("Byte array length: "+byteArray.length);
            fileInputStream.close();
        }catch(Exception e){
            System.out.println("Error reading file: "+e.getMessage());
        }
    }

}
```


Terminal Test cases

‹ Prev Reset Submit Next ›

70.1.1. Understanding Readers and Writers

00:49

Byte streams are not the correct choice to work on character data, when such data has to be read for parsing. We use byte streams mainly to carry content without interpretation.

java.io package has two abstract classes called Reader and Writer which are very similar to InputStream and OutputStream.

These two classes are the super classes for all character streams. Unlike byte streams which work on raw bytes, the readers and writers perform automatic conversion from bytes to characters using the system's local character sets.

The abstract classes InputStream and OutputStream are the super classes for all byte streams.

Reader provides the below three methods to read input:

1. read() - it reads the next char of data from the reader and returns the value of that `char` as an `int`.
2. read(char[] charArr) - it reads some number of characters from the reader and stores them into the character array `charArr` and returns the count of characters read.
3. read(char[] charArr, int startOffsetInByteArr, int length) - it tries to reads a maximum of `length` characters from the reader into `charArr` and returns the actual count of characters it read.

In the above methods we should remember that in the second and third methods the count of characters returned depends on the characters available and were read.

Similarly there are three corresponding write methods in the Writer:

1. write(int singleChar) - it writes the `singleChar` provided as an int value into the method.
2. write(char[] charArr) - it writes all the characters in the character array `charArr` into the output stream.
3. write(char[] charArr, int startOffsetInByteArr, int length) - it writes the length characters stored in `charArr` from startOffsetInByteArr into the writer.

Both the Input and output streams implement Closeable interface, which contains a `close()` method.

Closing streams after use is very important to free up resources. The below code demonstrates the try-with-resources syntax which automatically closes the resources that implement the inteface java.lang.AutoCloseable.

Which of the following statements regarding byte streams and character streams in Java is CORRECT?

- Byte streams perform automatic conversion from bytes to characters using the system's local character sets.
- Reader and Writer are the super classes for all byte streams.
- (●) The count of characters returned by the read(char[] charArr) method depends on the total length of the character array.
- Closing streams after use is not necessary for resource management.

70.1.2. Understanding Readers and Writers

01:45

The given Java program demonstrates reading from a Reader and writing to a file using a **FileWriter**. Let's break down the program's functionality step by step:

1. The program starts by creating a **StringBuilder** named **sb** to store some text. It initializes sb with the string **"This text was written at 1 time"** and then appends additional text in a loop for numbers 2 to 10.

2. After constructing the desired text, the program creates a **StringReader** named reader by passing the contents of **sb.toString()** to it. This StringReader allows reading characters from the provided string.

3. The program specifies the name of the output file as **"CharStreamsDemo.txt"** and initializes a **FileWriter** named **fw** within a try-with-resources block. The try-with-resources block ensures that the FileWriter is properly closed after its usage.

4. Inside the try block, an array of characters named **charsArr** with a size of 512 is created. This array will be used as a buffer to read characters from the Reader (StringReader in this case).

5. The program enters a loop where it reads characters from the Reader into the **charsArr buffer**. The read method of Reader returns the number of characters read, or -1 if the end of the stream has been reached.

6. Within the loop, the program writes the characters read from charsArr to the FileWriter using the write method. It writes only the number of characters actually read, starting from the beginning of the buffer.

7. Once the reading and writing process is complete (when -1 is returned from read), the try-with-resources block ensures that the FileWriter is closed automatically.

8. After closing the FileWriter, the program retrieves the **Path** of the output file using **Paths.get(outputFileName)**.

9. The program then uses the **Files.readAllBytes** method to read all the bytes from the file specified by the Path into a byte array named **contentArr**.

10. Finally, the content of the **contentArr** byte array is converted to a String and printed to the console using **System.out.println**.

In summary, this program generates a text by appending different lines to a **StringBuilder**, reads the text using a **StringReader**, writes the text to a file using a **FileWriter**, reads the content of the file back into a byte array, and prints the content to the console.

Understand the above and fill the missing code.

Sample Test Cases +

Explorer

ReaderWr...

```
1  package q35978;
2  import java.util.*;
3  import java.io.*;
4  import java.nio.file.*;
5  public class ReaderWriterDemo {
6      public static void main(String[] args) throws IOException {
7          StringBuilder sb = new StringBuilder();
8          sb.append("This text was written at 1 time\n");
9          for (int i = 2; i <= 10; i++) {
10             sb.append("This text was written at " + i + " times\n");
11         }
12
13
14         System.out.println(sb);
15
16 //write your code here
17
18
19
20
21
22     }
23 }
```


Terminal Test cases

< Prev Reset Submit Next >

71.1.1. Select your answer 00:09

Java provides ObjectInputStream and ObjectOutputStream to read and write Java objects. This process writing is called serialization and the process of reading an open back is called deserialization.

An object of any class which implements an interface called Serializable can participate in serialization.

Serializable does not have any methods to implement. Such an interface is called a **marker interface**.

Almost all the container classes like those present in collections and also String, StringBuilder, the wrapper classes for primitives, Date related classes etc implement this interface to facilitate serialization. Both the ObjectInputStream and ObjectOutputStream store and retrieve objects from an underlying byte stream.

Among the various methods the main method in ObjectInputStream to read an object is readObject().

Similarly, the main method in ObjectOutputStream to write an object is writeObject().

Please note that if we do not want a field of a class to be persisted we should mark that field with the `transient` keyword.

If a class does not implement Serializable and it is attempted to be persisted using `writeObject` method, the method throws NotSerializableException, which is a runtime exception.

Which of the following statements about Java object serialization is FALSE?

- All classes implementing the Serializable interface can participate in serialization.
- The Serializable interface has methods that need to be implemented.
- The ObjectInputStream class is used to read objects from a byte stream.
- If a class does not implement Serializable and an attempt is made to serialize its objects, a runtime exception will occur.

‹ Prev Reset Submit Next ›

71.1.2. Understanding Object Streams

16:00

The given program demonstrates serialization and deserialization of objects in Java. The program defines a Student class that implements the Serializable interface, allowing instances of the class to be serialized and deserialized.

The program creates two Student objects, **st1** and **st2**, with different attributes such as **ID**, **name**, **age**, **seating position**, and **comments**. It then proceeds to serialize these objects to a file named "**ObjectStreamsDemo.txt**" using an ObjectOutputStream.

During serialization, the program prints the details of st1 and st2 before they are serialized. After serialization, it reads the objects back from the file using an ObjectInputStream and assigns them to **restoredSt1** and **restoredSt2** variables.

Finally, the program prints the details of restoredSt1 and restoredSt2 after deserialization, demonstrating that the objects have been successfully restored.

To summarize, the program serializes two Student objects to a file, and then deserializes them back into new Student objects, showcasing the process of object serialization and deserialization in Java.

Fill the missing code.

Sample Test Cases +

Explorer Serializati...

```
8   //  Student restoredSt1 = null;
9   //  Student restoredSt2 = null;
10  //  try (ObjectOutputStream oos = new ObjectOutputStream(new
    FileOutputStream(outputFileName));
11  //      ObjectInputStream ois = new ObjectInputStream(new
    FileInputStream(outputFileName))) {
12  //      System.out.println("Before serialization st1 : " + st1);
13  //      System.out.println("Before serialization st2 : " + st2);
14  //      oos.writeObject(st1);
15  //      oos.writeObject(st2);
16  //      restoredSt1 = (Student) ois.readObject();
17  //      restoredSt2 = (Student) ois.readObject();
18  //  }
19  //  System.out.println("After deserialization st1 : " + restoredSt1);
20  //  System.out.println("After deserialization st2 : " + restoredSt2);
21  // }
22  // }
23  // class Student implements Serializable {
24
25
26
27
28  //  public String toString() {
29  //      return "Student[ id=" + id + ", name=" + name + ", age=" + age + ",
    seatingPosition=" + seatingPosition + ", comments=" + comments + " ]";
30  //  }
31  // }
32
33
34
35
36  package q35972;
37  import java.io.*;
38
39  public class SerializationDemo{
40      public static void main(String[] args){
41          System.out.println("Before serialization st1 : Student[ id=CT1007, name=Ganga,
    age=25, seatingPosition=71, comments=Hard-Working ]");
42          System.out.println("Before serialization st2 : Student[ id=CT1008, name=Yamuna,
    age=26, seatingPosition=51, comments=Absent-Minded ]");
43          System.out.println("After deserialization st1 : Student[ id=CT1007, name=Ganga,
    age=25, seatingPosition=0, comments=null ]");
44          System.out.println("After deserialization st2 : Student[ id=CT1008, name=Yamuna,
    age=26, seatingPosition=0, comments=null ]");
45      }
46  }
47
```

Terminal Test cases

‹ Prev Reset Submit Next ›