

Java Foundations

Exception Handling



Objectives

- This lesson covers the following objectives:
 - Explain the purpose of exception handling
 - Handle exceptions with a try/catch construct
 - Describe common exceptions thrown in Java



What Is an Exception?

- To understand exception handling, you need to first understand what is an exception
- An exception is an error that occurs during the execution of a program(run-time) that disrupts the normal flow of the Java program
- However, you can handle such conditions within your program and take necessary corrective actions so that the program can continue with its execution(exception handling)

Why Should You Handle Exceptions?

- If an exception occurs while your program is executing:
 - Execution of the program is terminated
 - A stack trace, with the details of the exception, is printed in the console

When You Don't Handle Exceptions: Example

- In Java, the following code throws an exception because you can't divide an integer by zero:

```
1 public class ExceptionHandling {  
2  
3     public static void main(String args[]) {  
4         int d = 0;  
5         int a = 10 / d; → Exception occurs at this statement  
6         System.out.print(a); → This statement isn't executed  
7     }//end method main  
8 } //end class ExceptionHandling
```

- A stack trace, with the details of the exception, is printed in the console
- Execution of the program is terminated at line 4, and so the statement at line 5 isn't executed

When You Don't Handle Exceptions

- When Java encounters an error or condition that prevents execution from proceeding normally, Java "throws" an exception
- If the exception isn't "caught" by the programmer, the program crashes
- The exception description and current stack trace are printed to the console

Dealing with Exceptions

- One way to deal with exceptions is to simply avoid them in the first place
- For example, avoid an `ArithmaticException` by using conditional logic:
 - Test to see if the condition will arise before you attempt the potentially risky operation

```
int divisor = 0;

if(divisor == 0){
    System.out.println("Can't be zero!");
}
else {
    System.out.println(5 / divisor);
}//endif
```

Exception Categories

- Java exceptions fall into two categories:
- Checked Exceptions:
 - Compiler checks and deals with exceptions
 - If the exceptions aren't handled in the program, it gives a compilation error
 - Examples:
 - `FileNotFoundException`, `IOException`
- Unchecked Exceptions:
 - Compiler does not check and deal with exceptions
 - Examples:
 - `ArrayIndexOutOfBoundsException`, `NullPointerException`, `ArithmaticException`



Exercise 1

- Import and open the ExceptionsEx project
- Examine ExceptionEx1.java:
 - Execute the program and observe the output:
 - ArrayIndexOutOfBoundsException occurs
 - Is it a good practice to handle the exception for this program?
 - Modify the program to compute the sum of the array

Handling Exceptions with the try/catch Block

- But not all exceptions can be prevented because you don't always know whether a given operation will fail before it's invoked
- Another strategy is to use the try/catch block for exception handling

Understanding the try/catch Block

- For code that's likely to cause an exception, you can write the code inside a special "try" block
- You associate exception handlers with a try block by providing one or more catch blocks after the try block
- Each catch block handles the type of exception indicated by its argument
- The `ExceptionType` argument type declares the type of exception

Flow Control in try/catch Blocks: Success

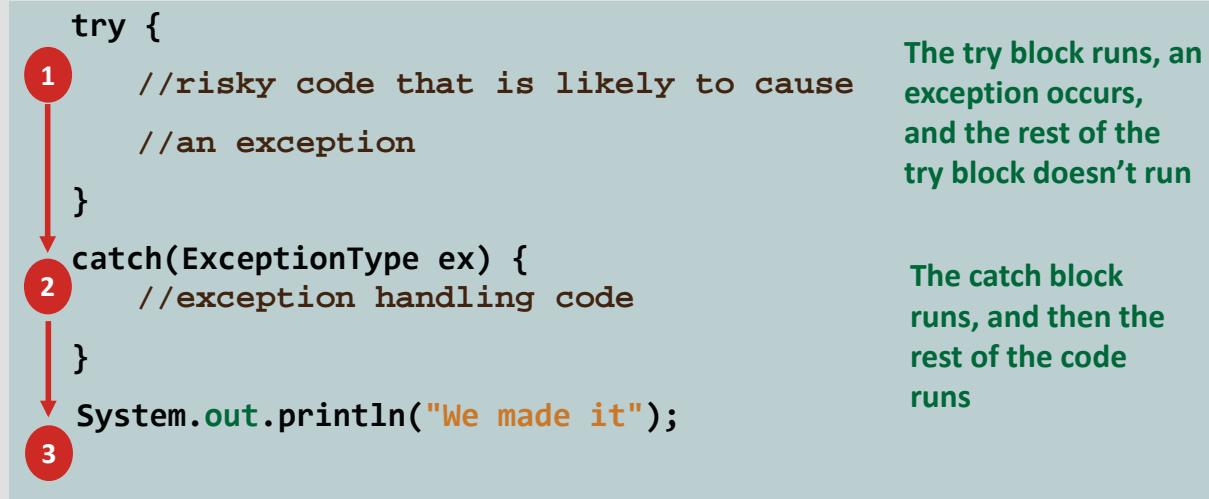
- If the try block succeeds, no exception occurs

```
try {  
    // risky code that is likely to cause  
    // an exception  
}  
  
catch(ExceptionType ex) {  
    // exception handling code  
}  
  
System.out.println("We made it");
```

First the try
block runs,
and then the
code after the
catch block
runs

Flow Control in try/catch Blocks: Failure

- If the try block fails, an exception occurs



Flow Control in try/catch Blocks: Example

```
1 public static void main(String args[]) {
2     int a = 100, res;
3     try{
4         System.out.println("Enter the value for b");
5         Scanner console = new Scanner(System.in);
6         int b = console.nextInt();
7         System.out.println("Enter the value for c");
8         int c = console.nextInt();
9         res = 10 / (b - c);
10        System.out.println("The result is " + res);
11    }
12    catch(Exception e){
13        String errMsg = e.getMessage();
14        System.out.println(errMsg);
15    }//end try catch
16    System.out.println("After catch block");
17 } //end method main
```

Examples of Exceptions

- `java.lang.ArrayIndexOutOfBoundsException`
 - Attempt to access a nonexistent array index
- `java.lang.NullPointerException`
 - Attempt to use an object reference that wasn't instantiated
- `java.io.IOException`
 - Failed or interrupted I/O operations

Understanding Common Exceptions

- Unchecked Exceptions - due to programming mistake :
 - Example:
 - `ArrayIndexOutOfBoundsException` exception

```
01 int[] intArray = new int[5];
02 intArray[5] = 27;
```

- Stack trace:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
    at TestErrors.main(TestErrors.java:17)
)
```

Identifying NullPointerException

- This unchecked exception is thrown when an application attempts to use null when an object is required
- These include:
 - Calling the instance method of a null object
 - Accessing or modifying the field of a null object

```
public static void main(String[] args) {  
  
    String name = null;  
    System.out.print("Length of the string " + name.length());  
  
}//end method main
```

Invoking the
length method
on a null object

Identifying IOException

```
public static void main(String[] args) {  
  
    try {  
        File testFile = new File("//testFile.txt");  
        testFile.createNewFile();  
        System.out.println("testFile exists:"  
                           + testFile.exists());  
    }  
    catch (IOException e) {  
        System.out.println(e);  
    }//end try catch  
}//end method main
```

Best Practices for Exception Handling

- Try to be as specific as possible with the type of error you're trying to catch
- This allows the program to provide you with specific feedback on what went wrong
- Catch a generic exception is often too imprecise to be useful, but can be done as a last resort

```
catch (Exception e) {  
    System.out.println(e);  
}
```

Example of Bad Practice

```
public static void main(String[] args) {  
  
    try {  
        File testFile = new File("//testFile.txt");  
        testFile.createNewFile();  
        System.out.println("testFile exists:  
                           " + testFile.exists());  
    }  
    catch (Exception e) {————— Catching any exception  
        System.out.println("Error Creating File");—————  
    } //end try catch  
} //end method main
```

No processing of
exception class?

Somewhat Better Practice

```
public static void main(String[] args) {  
    try {  
        File testFile = new File("//testFile.txt");  
        testFile.createNewFile();  
        System.out.println("testFile exists:"  
                           + testFile.exists());  
    }  
    catch (IOException e) {  
        System.out.println(e);  
    }//end try catch  
}//end method main
```

Catching specific exception
The `toString()` is called on this object

Exercise 2



- Import and open the ExceptionsEx project
- Examine Calculator.java and ShoppingCart.java
- Modify the programs to implement exception handling:
 - Calculator.java:
 - Identify the exception that might occur
 - Change the divide method signature to indicate that it throws an exception
 - ShoppingCart.java:
 - Catch the exception in the class that calls the divide method

Summary

- In this lesson, you should have learned how to:
 - Explain the purpose of exception handling
 - Handle exceptions with a try/catch construct
 - Describe common exceptions thrown in Java

