

I'm trying to teach my cat Java programming

But he keeps complaining about a
“`NullPointerException`”



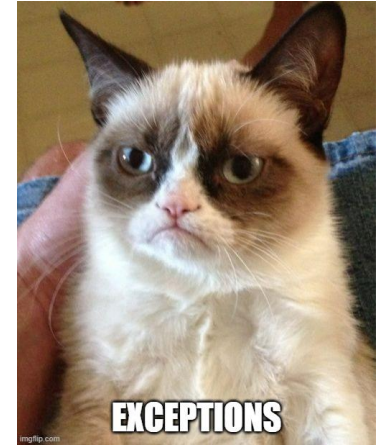


Module 1-16

Exceptions
File Input

Objectives

- Describe the concept of exception handling
- Implement a try/catch structure in a program
- Understand the java.io library File and Directory classes
- Explain what a character stream is
- Use a try-with-resources block
- Handle File I/O exceptions and how to recover from them
- Know how File I/O might be used on a job





Exceptions

What are Exceptions?

Exceptions are occurrences that alter the flow of the program away from the ideal or “happy” path.

- *Sometimes it's the developer's fault:* i.e. accessing an array element greater than the actual number of elements present.
- *Other times it's not:* i.e. loss of internet connection, a data file that was supposed to be there has been removed by a systems admin.

Checked vs. Unchecked Exceptions

- Checked are compile-time exceptions
 - If code in a method throws a checked exception, method must handle it
 - Handle in method or pass up to parent

```
File inputFile = getInputFileFromUser();
try(Scanner fileScanner = new Scanner(inputFile)) {
    while(fileScanner.hasNextLine()) {
        String line = fileScanner.nextLine();
        String rtn = line.substring(0, 9);

        if(checksumIsValid(rtn) == false) {
            System.out.println(line);
        }
    }
}
```

Unhandled exception type FileNotFoundException
2 quick fixes available:
1 Add throws declaration
2 Add catch clause to surrounding try

- Unchecked are run-time exceptions

- User or code does something that causes program to stop running

```
Cincinnati
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
at com.techelevator.exceptions.ExceptionsLecture.main(ExceptionsLecture.java:22)
```


Compile-time Exceptions (Checked Exceptions)

They are not runtime exceptions, but they must be handled or declared.

- **FileNotFoundException:** This is thrown programmatically, when the program tries to do something with a file that doesn't exist.
- **IOException:** A more general exception related to problems reading or writing to a file.
 - Note that `FileNotFoundException` extends from `IOException`.

```
File inputFile = getInputFileFromUser();
try(Scanner fileScanner = new Scanner(inputFile)) {
    while(fileScanner.hasNextLine()) {
        String line = fileScanner.nextLine();
        String rtn = line.substring(0, 9);

        if(checksumIsValid(rtn) == false) {
```



The image shows a code editor snippet with a Java try-with-resources block. The code is not yet closed with a closing curly brace. An IDE error tooltip is visible on the right side of the code, indicating an 'Unhandled exception type FileNotFoundException'. The tooltip lists two quick fixes: 'Add throws declaration' and 'Add catch clause to surrounding try'. The code snippet is as follows:

Runtime Exceptions (Unchecked Exceptions)

Runtime exceptions are errors that occur whilst the program is executing in the JVM. Here are three common examples:

- **NullPointerException**: you tried to call a method or access a data member for a null reference.
- **ArithmeticException**: you tried to divide by zero.
- **ArrayIndexOutOfBoundsException**: you tried to access an array element with an index that is out of bounds.

Exceptions “Throwing”

Throwing means making everyone aware that a deviation from normal program flow has occurred.

- Throwing can be done behind the scenes by the JVM.
- It can be triggered via code, by using the *throw* statement.



Exceptions “Handling”

Handling are the actions taken (defined by the programmer) when an exception is encountered.



Java exceptions in a nutshell

Try / Catch

The Try Catch block follows the following format:

```
try {  
    // Code where an exception might be triggered  
}  
catch (FileNotFoundException e) {  
    // Catch and specify actions to take if an exception is encountered.  
}  
finally {  
    // Action to take regardless of whether an exception was encountered.  
}
```

Both the catch and finally blocks are optional but one of them must be present (either try or finally, or both).

Try / Catch

```
16 System.out.println("The following cities: ");
17 String[] cities = new String[] { "Cleveland", "Columbus", "Cincinatti" };
18 try {
19     System.out.println(cities[0]);
20     System.out.println(cities[1]);
21     System.out.println(cities[2]);
22     System.out.println(cities[3]); // This statement will throw an ArrayIndexOutOfBoundsException
23     System.out.println("are all in Ohio."); // This line won't execute because the previous statement throws an Exception
24 } catch (ArrayIndexOutOfBoundsException e) {
25     // Flow of control resumes here after the Exception is thrown
26     System.out.println("XXX Uh-oh, something went wrong... XXX");
27 }
28
```

Let's throw some exceptions!

Exceptions Handling: Example

Consider the following example:

```
import java.io.FileNotFoundException;

public class SuspiciousClass {

    public void doSomething() throws
        FileNotFoundException {

        throw new FileNotFoundException();
    }
}
```

An exception is
programmatically thrown.

```
public class MyMainClass {

    public static void main(String[] args) {
        SuspiciousClass test = new
            SuspiciousClass();
        test.doSomething();
    }
}
```

Java will complain as we try
to invoke doSomething() as it
expects us to handle or catch
the exception.

Exceptions Handling: Example

Our first choice is to just state that on the main method (from which we call doSomething) that there is a possibility an exception will be thrown:

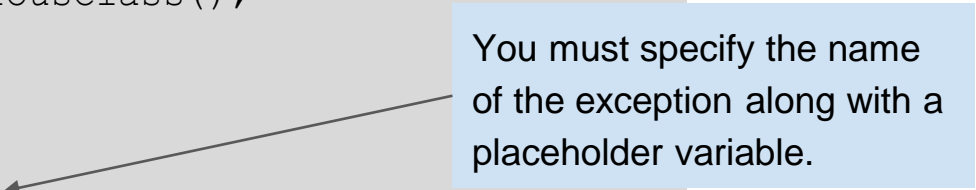
```
public static void main(String[] args) throws  
    FileNotFoundException {  
  
    SuspiciousClass test = new SuspiciousClass();  
    test.doSomething();  
  
}
```

Exceptions Handling: Example

Or, we could use a try / catch block to both catch the exception and specify a set of actions to do in the event we run into the exception.

```
public static void main(String[] args) {  
  
    SuspiciousClass test = new SuspiciousClass();  
  
    try {  
        test.doSomething();  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("ok... that's fine, moving on.");  
    }  
}
```

You must specify the name of the exception along with a placeholder variable.



File Input



File Input

Java has the ability to read in data stored in a text file.

It is one of many forms of inputs available to Java:

- Command Line user input
(we have covered this one)
- Through a relational database
(Module 2)
- Through an external API
(Module 2)



APCS: "Find the average of this .txt file full of integers using an array" Me:

File Input : The File Class

The file class is the Java class that encapsulates what it means to be a file containing data. This is an instantiation of a File object.

```
File <<variable name>> = new File(<<Location of the file>>);
```

In its simplest form it has a constructor that takes in the location of the file (including the name). Here is a concrete example:

```
File inputFile = new File("testFile.txt");
```

File Input : The File Class

The file location corresponds to the root of that particular Java project. Again, in this example our file is testFile.txt:

Name

- .idea
- src
- target
- m01d16-file-io-p
- pom.xml
- rtn.txt

In this example, testFile.txt is located in the project root, we can refer to it like so:

```
File inputFile = new  
    File("testFile.txt");
```

Name

- .idea
- resources
- src
- target
- m01d16-file-io-part1-lect
- pom.xml

In this example, testFile.txt has been moved **inside a folder called resources**.

```
File inputFile = new  
File("resources/testFile.txt");
```

File Input : The File Class Methods

There are several methods of the file class that can be used for file input:

- **.exists()**: returns a boolean to check to see if a file exists. We would not want to proceed to parse a file if the file itself was missing!
- **.isFile()**: returns a boolean to check to see if what we are looking at is a File. Returns false if it is not a file (perhaps a folder)
- **.getAbsolutePath()**: returns the same File object you instantiated but with an absolute path. You can think of this as a getter. It returns a File object.

File and Scanner

A File object and a Scanner object will work in conjunction with one another to read the file data.

Once a file object exists, we instantiate a Scanner object with the file as a constructor argument. Previously, we used System.in as the argument.



File and Scanner: Example

Consider this example:

```
public static void main(String[] args) throws FileNotFoundException {  
  
    File inputFile = new File("resources/testFile.txt");  
  
    if (inputFile.exists()) {  
        System.out.println("found the file");  
    }  
  
    try (Scanner inputScanner = new Scanner(inputFile)) {  
  
        while (inputScanner.hasNextLine()) {  
            String lineInput = inputScanner.nextLine();  
            String [] wordsOnLine = lineInput.split(" ");  
  
            for (String word : wordsOnLine) {  
                System.out.print(word + ">>>");  
            }  
        }  
    }  
}
```

We need to handle an exception, but we can pass it up to the parent class.

New file object being instantiated.

Instantiating a scanner but using a file object instead of System.in.

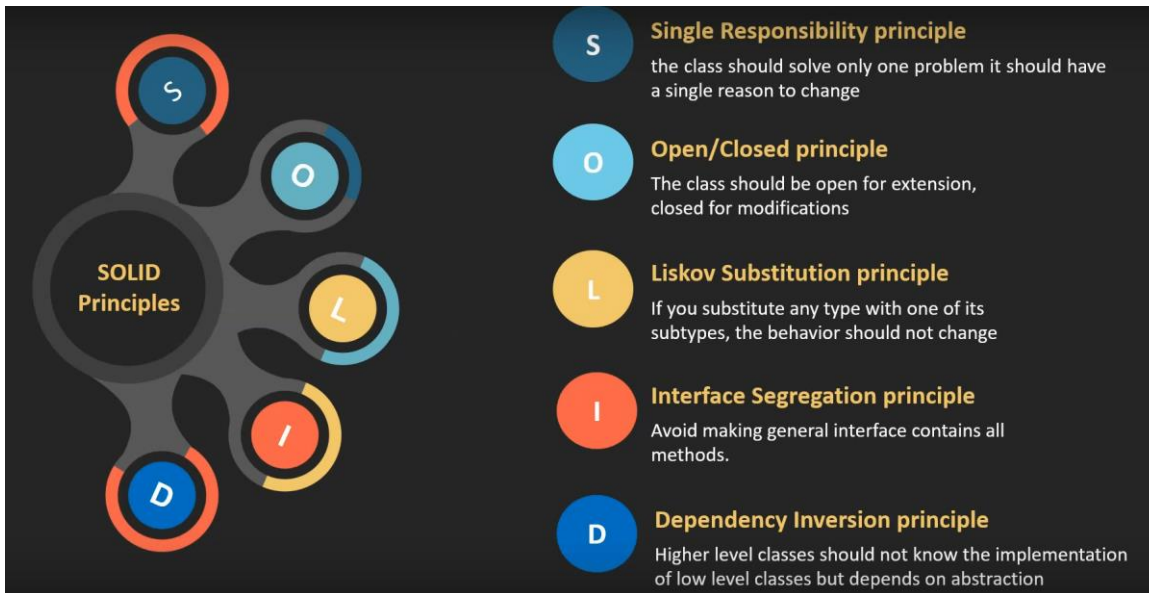
The while loop will iterate until it has processed all lines.

File and Scanner: Example

Here is a cleaner version of the example:

```
public static void main(String[] args) throws FileNotFoundException {  
  
    File inputFile = new File("resources/testFile.txt");  
  
    if (inputFile.exists())  
        System.out.println("found the file");  
    }  
  
    try (Scanner inputScanner = new Scanner(inputFile)) {  
  
        while (inputScanner.hasNextLine()) {  
            String lineInput = inputScanner.nextLine();  
            String [] wordsOnLine = lineInput.split(" ");  
  
            for (String word : wordsOnLine) {  
                System.out.print(word + ">>>");  
            }  
        }  
    }  
}
```


SOLID Principles



<https://www.educative.io/edpresso/what-are-the-solid-principles-in-java>

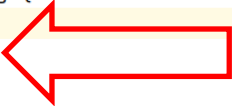
<https://www.jrebel.com/blog/solid-principles-in-java>

SOLID Principles

- SRP – Single Responsibility Principle
 - Every class (or similar structure) should only have one job to do

```
// abstract class means cannot create a concrete object
public abstract class Dwelling {
    |
    private String type;
    private String address;
    private int squareFootage;
    private int numberOfBedrooms;
    private double numberOfBathrooms;
    private BigDecimal monthlyPrice;
    private BigDecimal deposit;

    public Dwelling(String type, String address, int squareFootage,
                    int numberOfBedrooms, double numberOfBathrooms,
                    BigDecimal monthlyPrice, BigDecimal deposit) {
        this.type = type; // House, Condo, or Apartment
        this.address = address;
        this.squareFootage = squareFootage;
        this.numberOfBedrooms = numberOfBedrooms;
        this.numberOfBathrooms = numberOfBathrooms;
        this.monthlyPrice = monthlyPrice;
        this.deposit = deposit;
    }
}
```



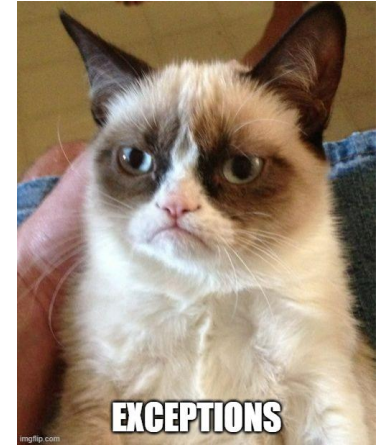
SOLID Principles

- SRP – Single Responsibility Principle
 - Every class (or similar structure) should only have one job to do
- OCP – Open Closed Principle
 - Classes should be open for extension but closed for modification
- LSP – Liskov Substitution Principle
 - In inheritance, design your classes so that dependencies can be substituted without needing modification in the client (use interfaces)
 - If it looks like a Duck, quacks like a Duck, but needs batteries, you probable have the wrong extraction (Tractor was not a child of FarmAnimal)
- ISP – Interface Segregation Principle
 - Keep interfaces small so you don't force classes to provide methods that have no meaning
- DIP – Dependency Inversion Principle
 - High-level modules should not depend on low-level modules, they should depend on abstractions

<https://www.jrebel.com/blog/solid-principles-in-java>

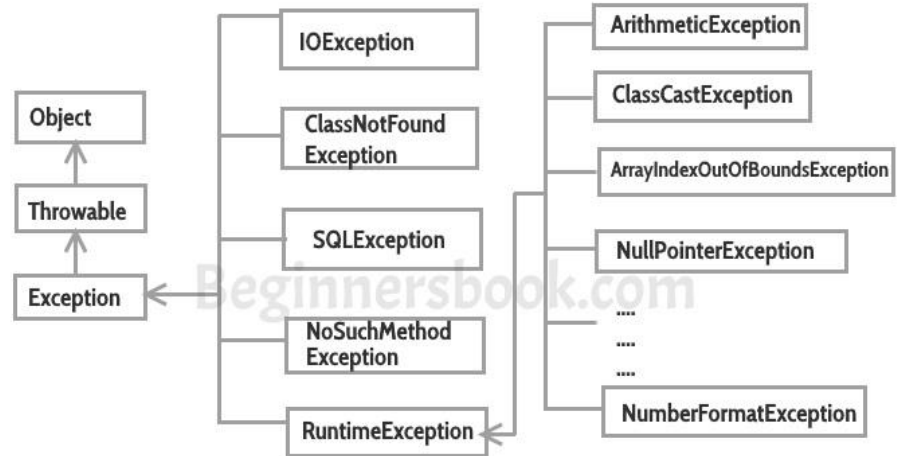
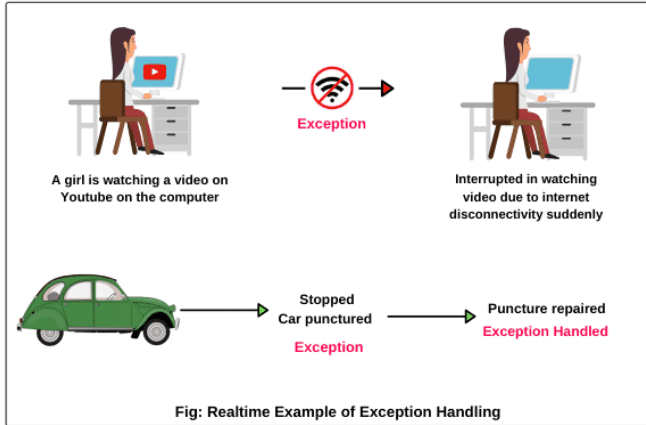
Objectives

- Describe the concept of exception handling
- Implement a try/catch structure in a program
- Understand the java.io library File and Directory classes
- Explain what a character stream is
- Use a try-with-resources block
- Handle File I/O exceptions and how to recover from them
- Know how File I/O might be used on a job



Objectives

- Describe the concept of exception handling



Objectives

- Describe the concept of exception handling
- Implement a try/catch structure in a program

```
try {
    try {
        int result = 1 / 0;
    } catch (SomeException e) {
        System.out.println("Something caught");
    } finally {
        System.out.println("Not quite finally");
    }
} catch (ArithmeticException e) {
    System.out.println("ArithmeticException caught");
} finally {
    System.out.println("Finally");
}
```

```
try {
    foo(10);
} catch (Exception ie) {
    System.out.println(ie.getMessage());
} catch (NullPointerException ne) {
    System.out.println(ne.getMessage());
}
```

Unreachable catch block for NullPointerException. It is already handled by the catch block for Exception

2 quick fixes available:

- [Remove catch clause](#)
- [Replace catch clause with throws](#)

Objectives

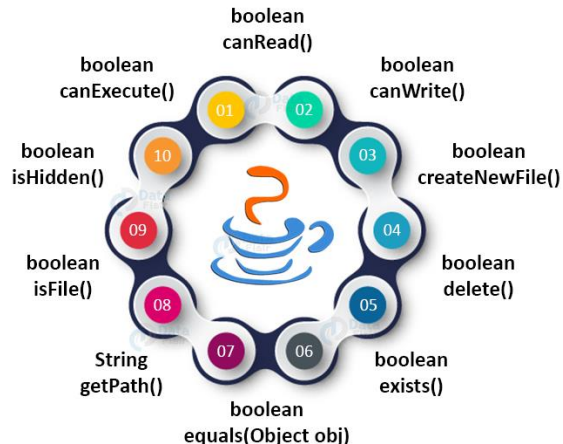
- Describe the concept of exception handling
- Implement a try/catch structure in a program
- Understand the java.io library File and Directory classes

```
AbsoluteAndCanonicalPathExample.java
1 package com.journaldev.examples;
2
3 import java.io.File;
4
5 public class AbsoluteAndCanonicalPathExample {
6
7     public static void main(String[] args) throws IOException {
8         File file = new File("/Users/pankaj/source.txt");
9         File file1 = new File("/Users/pankaj/temp/./source.txt");
10
11         System.out.println("Absolute Path : " + file.getAbsolutePath());
12         System.out.println("Canonical Path : " + file.getCanonicalPath());
13
14         System.out.println("Absolute Path : " + file1.getAbsolutePath());
15         System.out.println("Canonical Path : " + file1.getCanonicalPath());
16     }
17 }
18
19
20 }
```

Problems Javadoc Declaration Search Console Progress Call Hierarchy

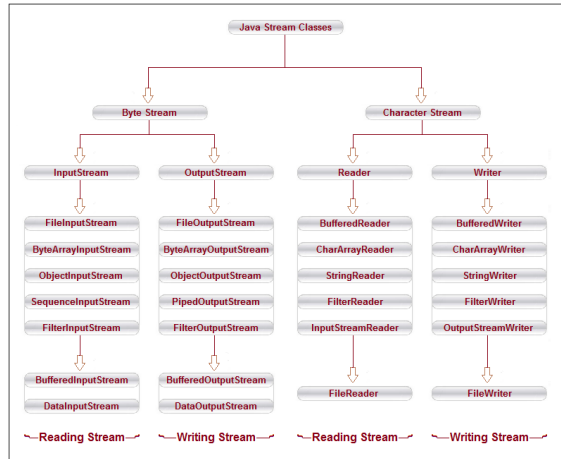
<terminated> AbsoluteAndCanonicalPathExample (1) [Java Application] /Library/Java/JavaVirtualMachines/...
Absolute Path : /Users/pankaj/source.txt
Canonical Path : /Users/pankaj/source.txt
Absolute Path : /Users/pankaj/temp/./source.txt
Canonical Path : /Users/pankaj/source.txt

Methods of File Class in Java



Objectives

- Describe the concept of exception handling
- Implement a try/catch structure in a program
- Understand the java.io library File and Directory classes
- Explain what a character stream is



```
package com.lynda.javatraining.characterstreams;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        try {
            FileReader in = new FileReader("textfile.txt");
            FileOutputStream out = new FileOutputStream("newfile.txt");
        } {
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


Objectives

- Describe the concept of exception handling
- Implement a try/catch structure in a program
- Understand the java.io library File and Directory classes
- Explain what a character stream is
- Use a try-with-resources block

```
14 try(FileReader fr = new FileReader("pop.txt")){
15     System.out.println("Reading from file");
16     int c1 = fr.read();
17     while (c1 != -1) {
18         System.out.print((char) c1);
19         c1 = fr.read();
20     }
21 } catch (FileNotFoundException e1) {
22     e1.printStackTrace();
23 } catch (IOException e) {
24     e.printStackTrace();
25 }
26
```

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

class Main {

    public static void main(String[] args) throws IOException{
        try (Scanner scanner = new Scanner(new File("testRead.txt"));
            PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {
            while (scanner.hasNext()) {
                writer.print(scanner.nextLine());
            }
        }
    }
}
```

Objectives

- Describe the concept of exception handling
- Implement a try/catch structure in a program
- Understand the java.io library File and Directory classes
- Explain what a character stream is
- Use a try-with-resources block
- Handle File I/O exceptions and how to recover from them

IOException

```
01 import java.io.FileInputStream;
02 import java.io.FileNotFoundException;
03 public class FileNotFoundExceptionExample
04 {
05     public void checkFileNotFound()
06     {
07         try
08         {
09             FileInputStream in = new FileInputStream("input.txt");
10             System.out.println("This is not printed");
11         }
12         catch (FileNotFoundException fileNotFoundException)
13         {
14             fileNotFoundException.printStackTrace();
15         }
16     }
17     public static void main(String[] args)
18     {
19         FileNotFoundExceptionExample example = new FileNotFoundExceptionExample();
20         example.checkFileNotFound();
21     }
22 }
```

The code above is executed as shown below:

Run Command

```
1 javac InputOutputExceptionExample.java
2 java InputOutputExceptionExample
```

Objectives

- Describe the concept of exception handling
- Implement a try/catch structure in a program
- Understand the java.io library File and Directory classes
- Explain what a character stream is
- Use a try-with-resources block
- Handle File I/O exceptions and how to recover from them
- Know how File I/O might be used on a job

