Session 6 (of 24)

PGR112 Objektorientert programmering

Yuan Lin / yuan.lin@kristiania.no

Today’s Main Goal

- Exception Handling
- Read from File
- Write to File
- Logger
- Practice Time

Status

- Step 1: IntelliJ, Git, Hello World
- Step 2: Variable, Method, Data type, Scope, Control statement
- Step 3: Class, Object, Scanner, Encapsulation
- Step 4: Package, Import, Modifiers, Overload, ArrayList, Enums
- Step 5: Scanner (cont.), Input validation, Debugger, Write test
- Now: Exception handling, File handler, Logger
- …

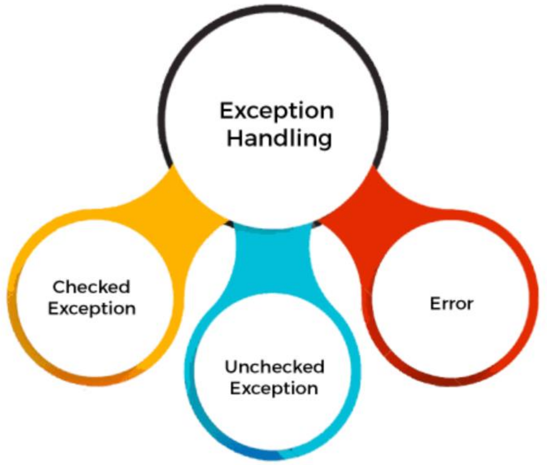
What is Exception?

- What is Exception in Java?
  - In Running programs, situations may arise where something abnormal is happening.
  - “Abnormal” means that the code does not run as it is intended to.
  - Java definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program’s instructions. It is an object which is thrown at runtime.

- An exception can occur for various reasons
  - A user has entered an invalid data
  - File not found
  - A network connection has been lost in the middle of communications
  - The JVM has run out of memory

Examples of Exception

- Type of Java Exceptions
  - According to Oracle, there are three types of exceptions namely
    - Checked Exception
    - Unchecked Exception
    - Error
  - Error is considered as unchecked exceptions
- Error vs Exception
  - Error indicates a serious problem that application should not try to catch
  - Exception indicates conditions that application might try to catch.

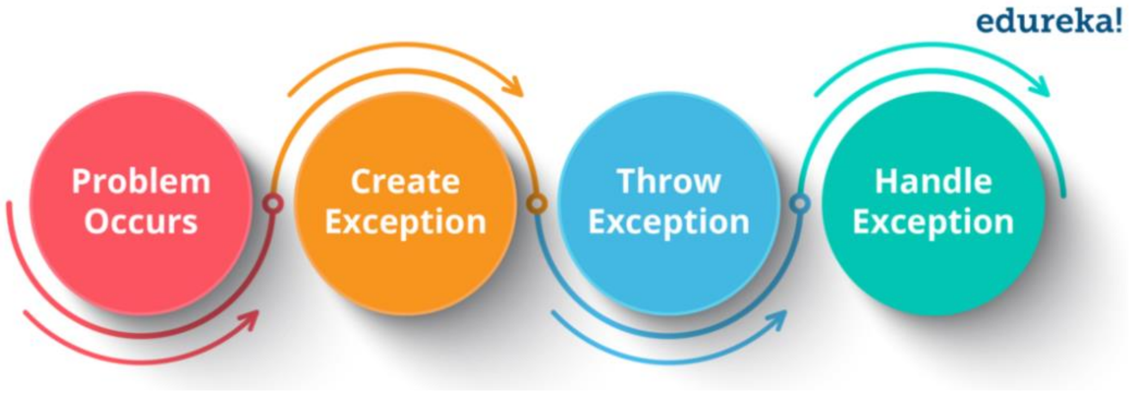


Src: javatpoint

When an Exception Occurs

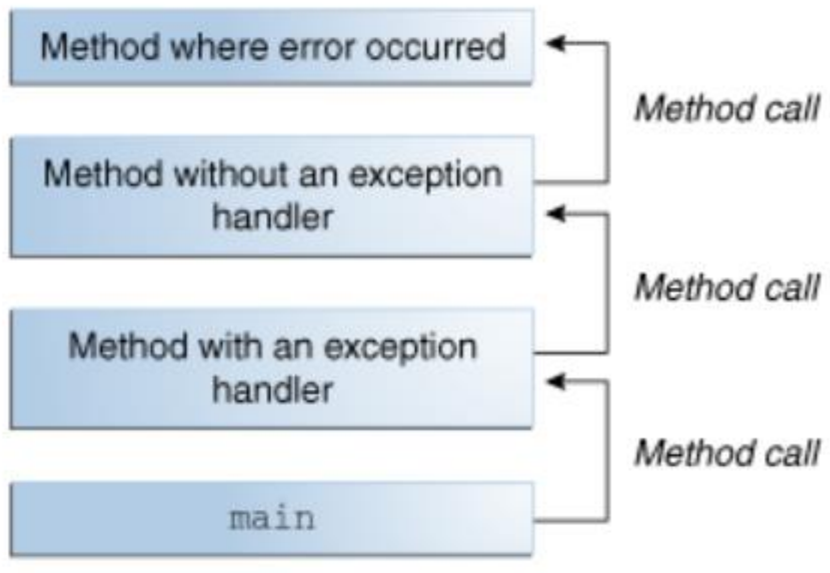
- When an exception occurs
  - Error occurs inside a method
  - Creates Exception: an object, contains information about the error, including the type and condition of the program, is created
  - Throw exception: the exception object is delivered to the runtime system
  - Handle exception: The runtime system tries to find something to handle the exception

- The “something” is the ordered list of methods that had been called to get to the method where the error occured.
- The list of methods is called “Call Stack”

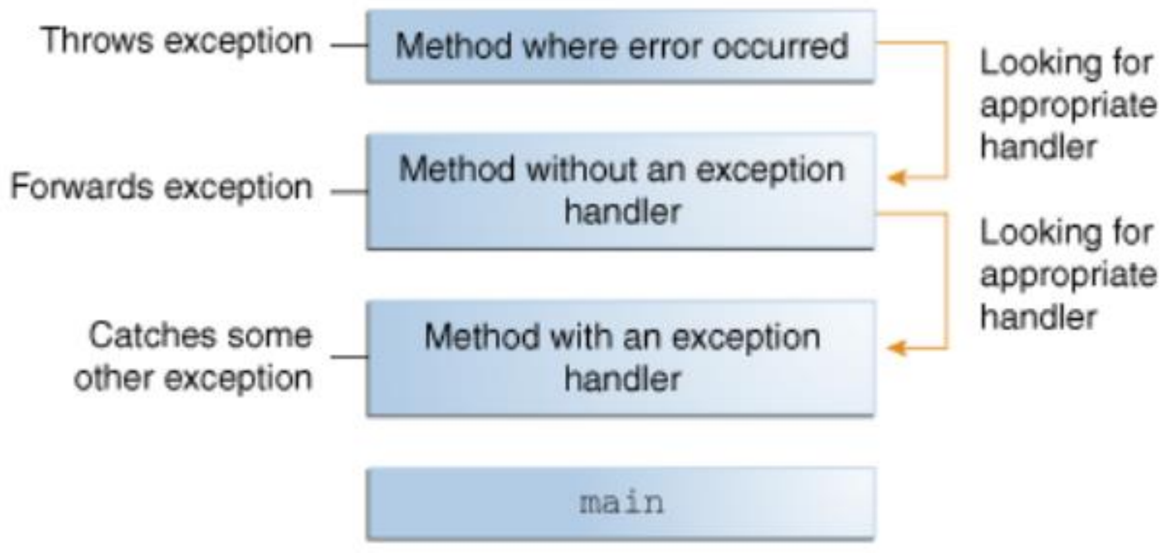


Src: https://medium.com/edureka/java-exception-handling-7bd07435508c

Call Stack


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

The call stack.

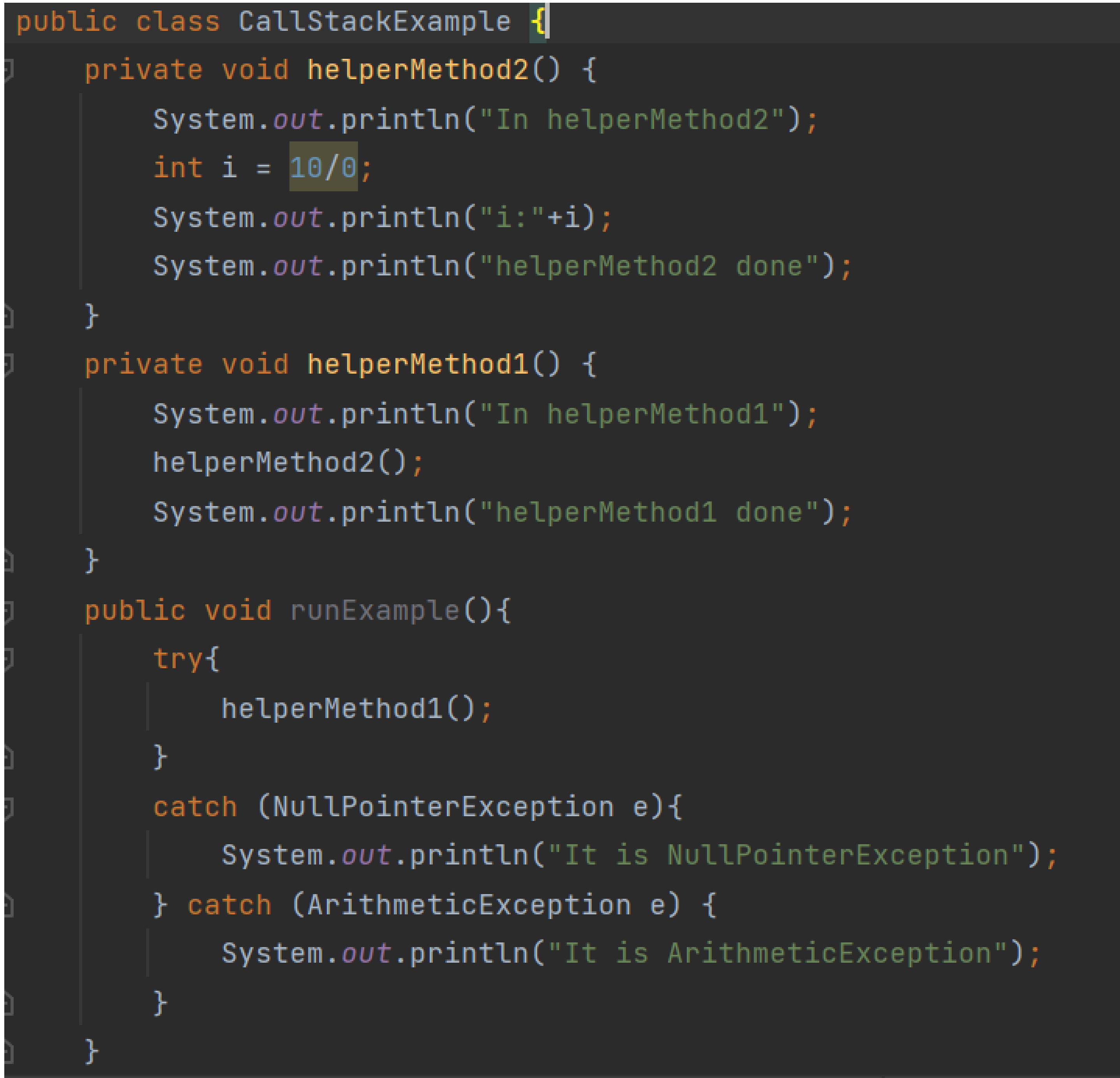

Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

Searching the call stack for the exception handler.

Src: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Call stack and exception handling

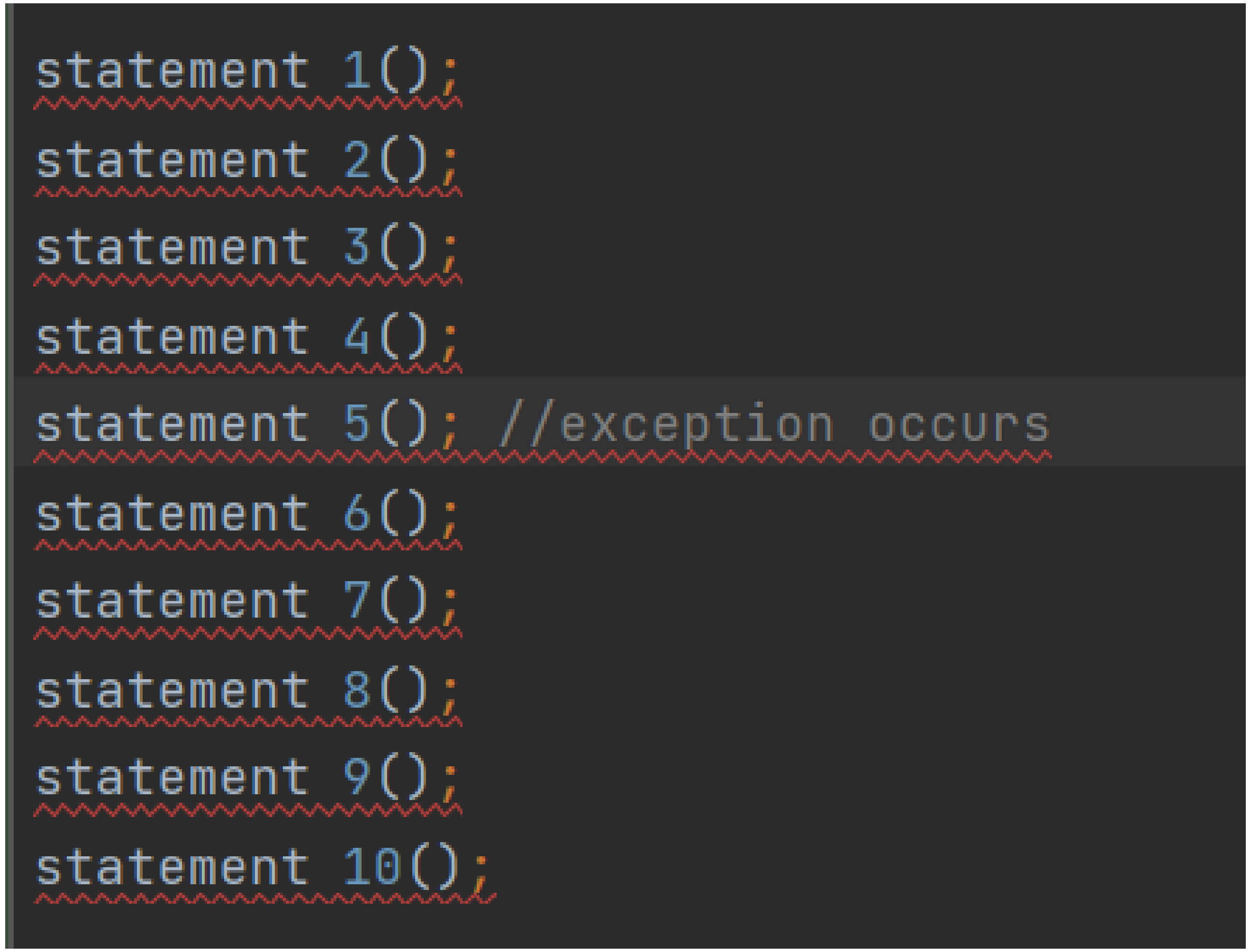
- The core advantage of exception handling is to maintain the normal flow of the application.
- Other advantages of exception handling include…
- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system terminates.



```
public class CallStackExample {
    private void helperMethod2() {
        System.out.println("In helperMethod2");
        int i = 10/0;
        System.out.println("i:"+i);
        System.out.println("helperMethod2 done");
    }
    private void helperMethod1() {
        System.out.println("In helperMethod1");
        helperMethod2();
        System.out.println("helperMethod1 done");
    }
    public void runExample(){
        try{
            helperMethod1();
        }
        catch (NullPointerException e){
            System.out.println("It is NullPointerException");
        } catch (ArithmeticException e) {
            System.out.println("It is ArithmeticException");
        }
    }
```

What is Exception Handling?

- Exception handling is a mechanism to handle runtime errors.
- Consider below scenario:
  - There are 10 statements in a Java program.
  - An exception occurs in statement 5.
  - Without exception handling, statements 6 -10 will not be executed.
  - When we perform exception handling, the rest of the statements will be executed.



```
statement 1();
statement 2();
statement 3();
statement 4();
statement 5(); //exception occurs
statement 6();
statement 7();
statement 8();
statement 9();
statement 10();
```

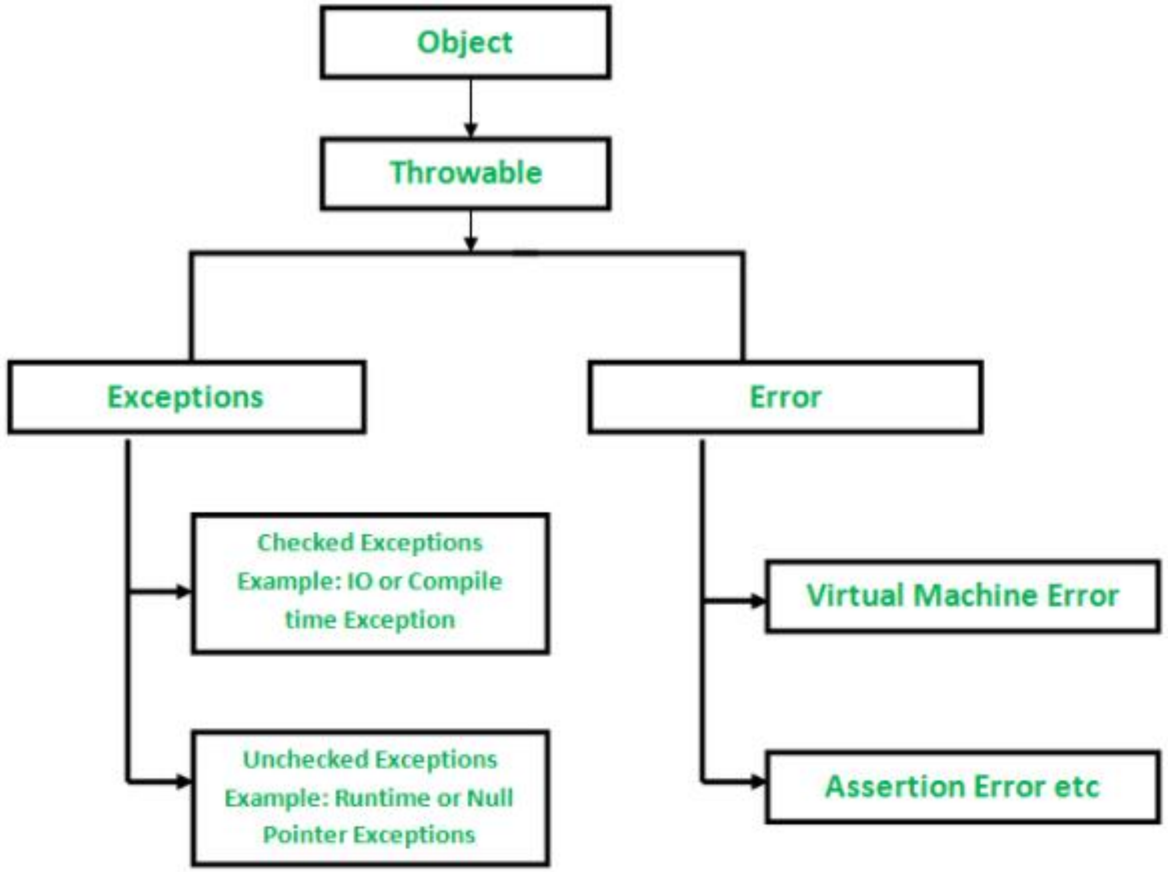
Three main groups of exceptions

- There are three main groups of exceptions
  - Unchecked exceptions. Bugs or logical errors. Programmer's error
  - Checked exceptions. Exceptions checked at compile time
  - Errors (The application should not try to catch. i.e., hardware failure)
- Unchecked exceptions are not necessarily to be handled sicne compiler does not check
- Checked exceptions must be handled <catch or specify>
  - try-catch clause, or
  - Throws the excepton further – how? Using throws method signature
- Exception class is the superclass of all exception classes in Java

Example of Checked vs Unchecked Exceptions

- Checked exception (relevant to this week especially)
  - FileNotFoundException
  - IOException

- Unchecked exception
  - ArithmeticException
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - NumberFOrmatException



Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Src: https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

Catch or Specify

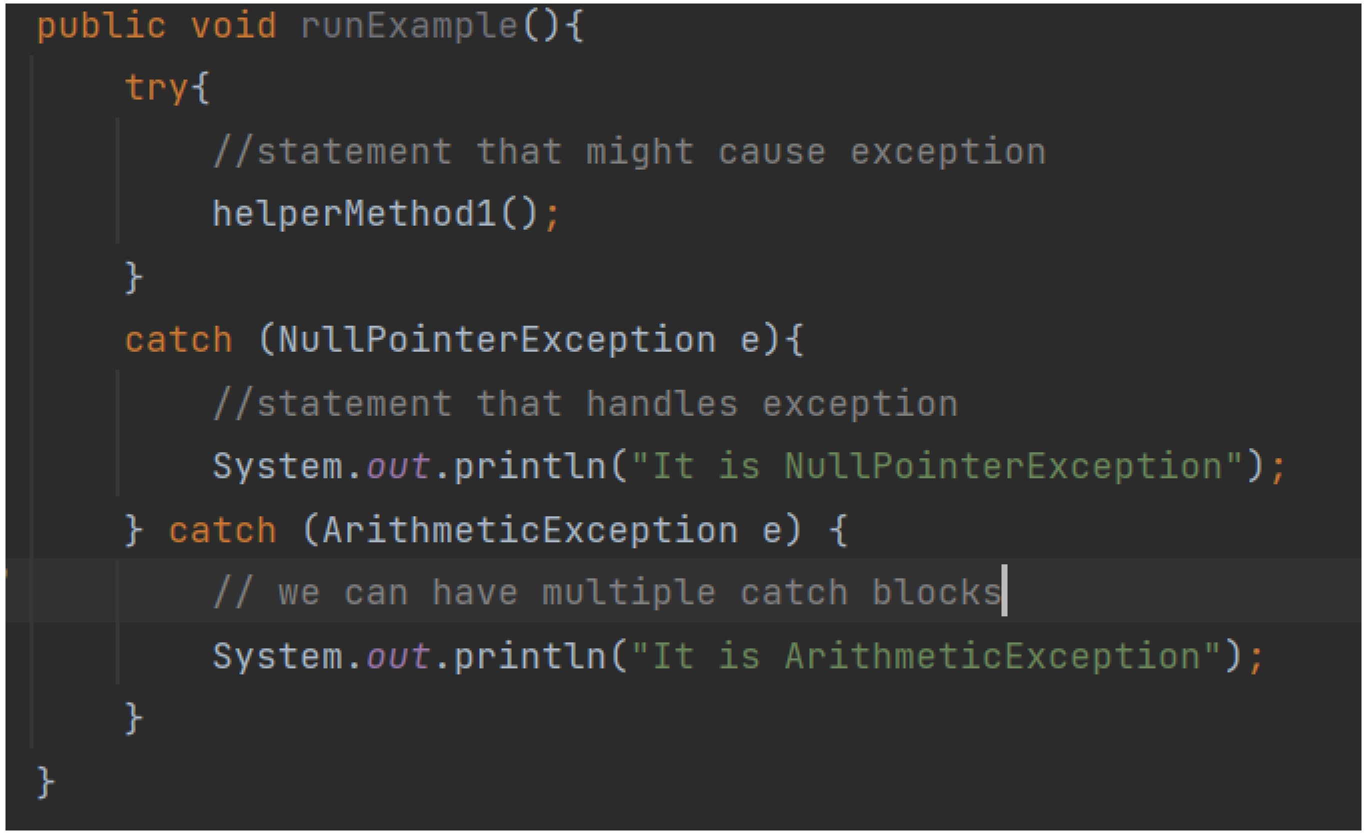
- The main difference between checked and unchecked exceptions is that we MUST take checked exceptions into account
- Because the compiler will check

- So how do we handle a checked exception?
  - Try/catch block
  - Throw the exception further using throws clause (it is called method signature)

Try-catch block

- **Try** specifies a block where we should place an exception code

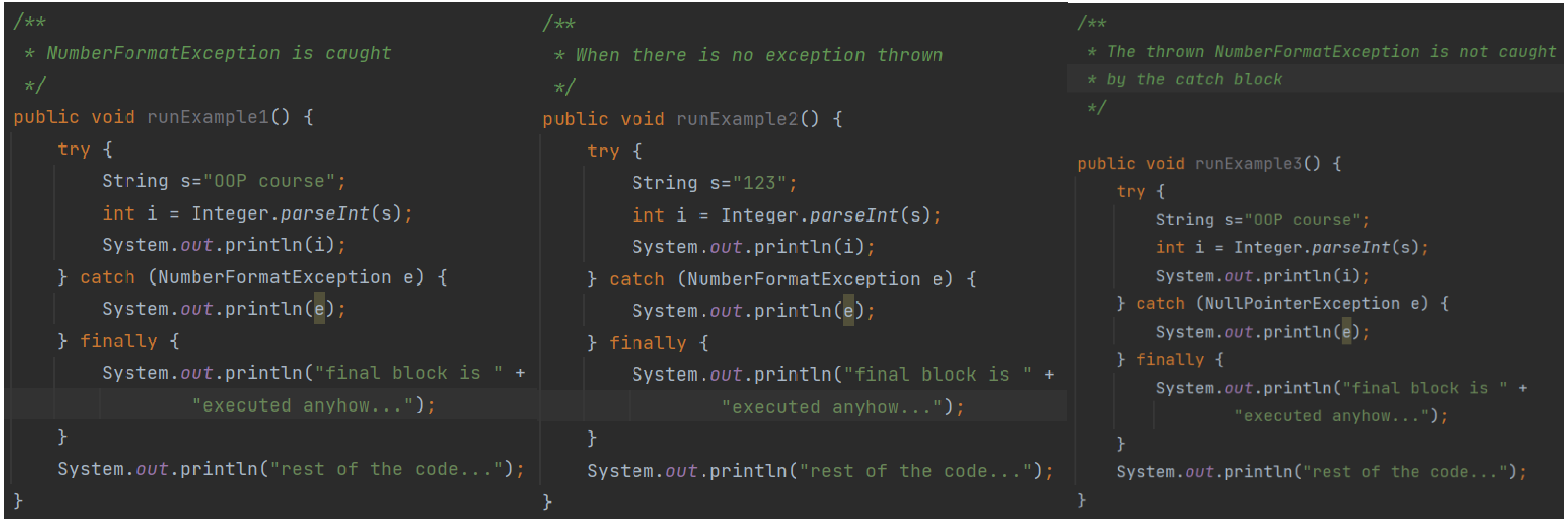
- **Catch** specifies a block where exception handling locates in.



```
public void runExample(){
    try{
        //statement that might cause exception
        helperMethod1();
    }
    catch (NullPointerException e){
        //statement that handles exception
        System.out.println("It is NullPointerException");
    } catch (ArithmeticException e) {
        // we can have multiple catch blocks
        System.out.println("It is ArithmeticException");
    }
}
```

Finally

- In addition to try/catch, we can have a finally block that runs after catch.
- Finally block will be executed in any circumstances as shown below
- We usually put cleaningup code in finally block



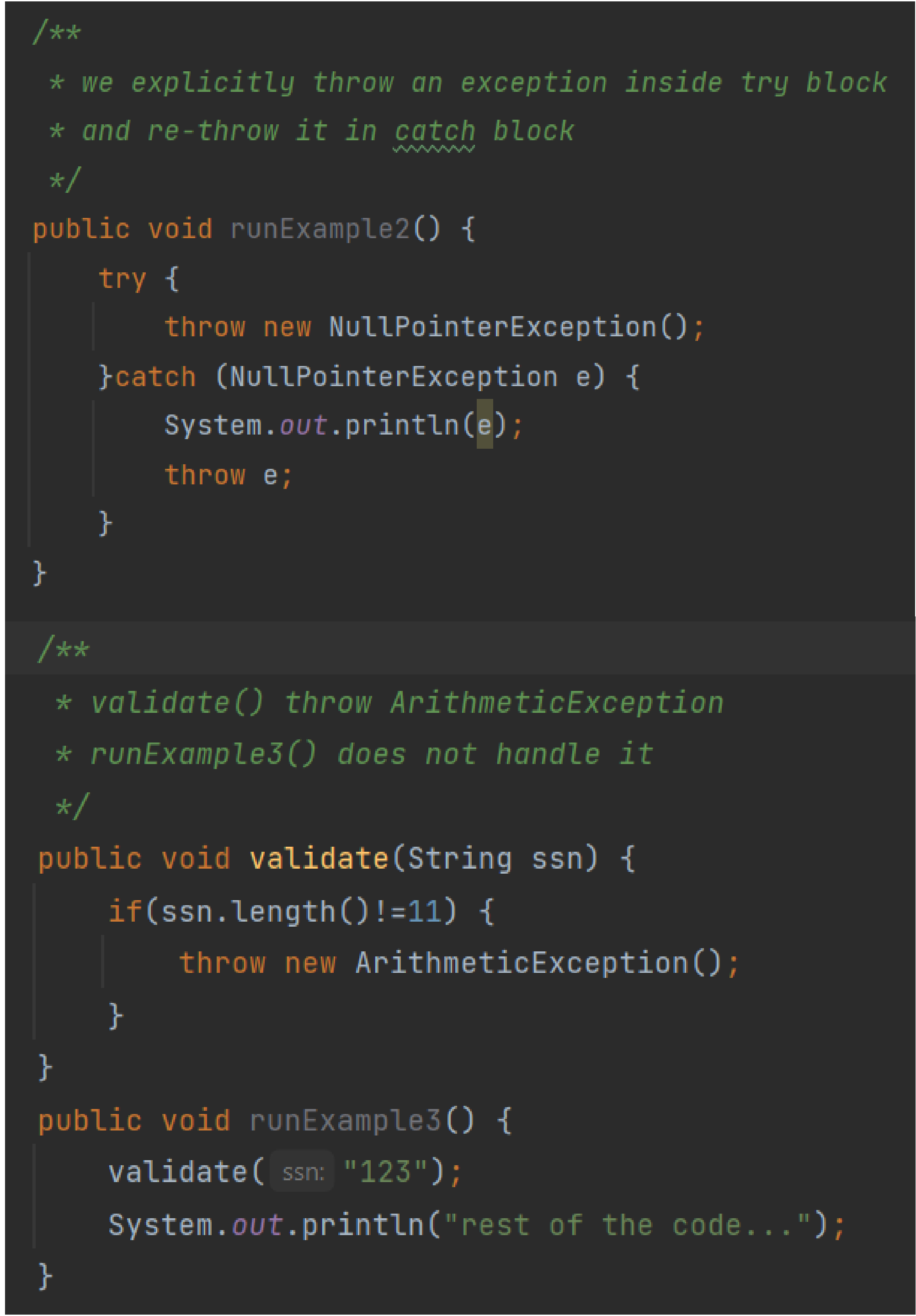
```java
/**
 * NumberFormatException is caught
 */
public void runExample1() {
    try {
        String s="OOP course";
        int i = Integer.parseInt(s);
        System.out.println(i);
    } catch (NumberFormatException e) {
        System.out.println(e);
    } finally {
        System.out.println("final block is " +
                "executed anyhow...");
    }
    System.out.println("rest of the code...");
}
```

```java
/**
 * When there is no exception thrown
 */
public void runExample2() {
    try {
        String s="123";
        int i = Integer.parseInt(s);
        System.out.println(i);
    } catch (NumberFormatException e) {
        System.out.println(e);
    } finally {
        System.out.println("final block is " +
                "executed anyhow...");
    }
    System.out.println("rest of the code...");
}
```

```java
/**
 * The thrown NumberFormatException is not caught
 * by the catch block
 */

public void runExample3() {
    try {
        String s="OOP course";
        int i = Integer.parseInt(s);
        System.out.println(i);
    } catch (NullPointerException e) {
        System.out.println(e);
    } finally {
        System.out.println("final block is " +
                "executed anyhow...");
    }
    System.out.println("rest of the code...");
}
```

Throw

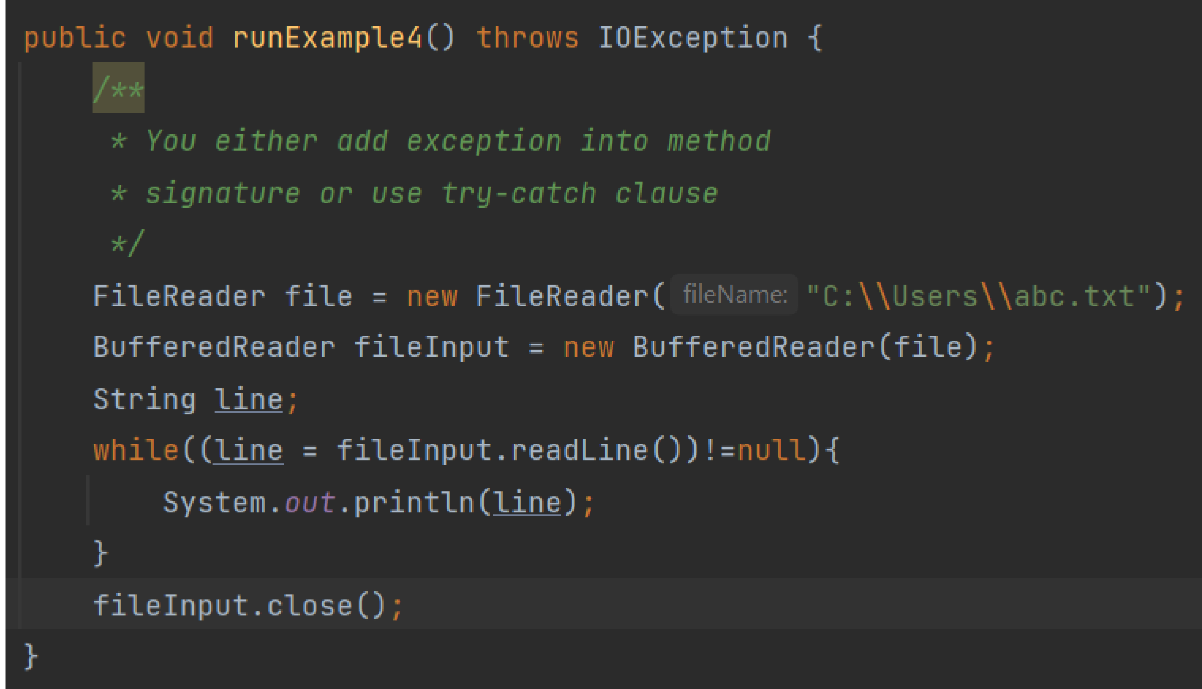
- The throw keyword is used to throw an exception explicitly within a method
- We can throw either checked or unchecked exceptions
- We can re-throw an exception from catch block
- Throw keyword is mainly for throwing a custom exception
- Syntax: The throw keyword is followed by an instance of Exception to be thrown



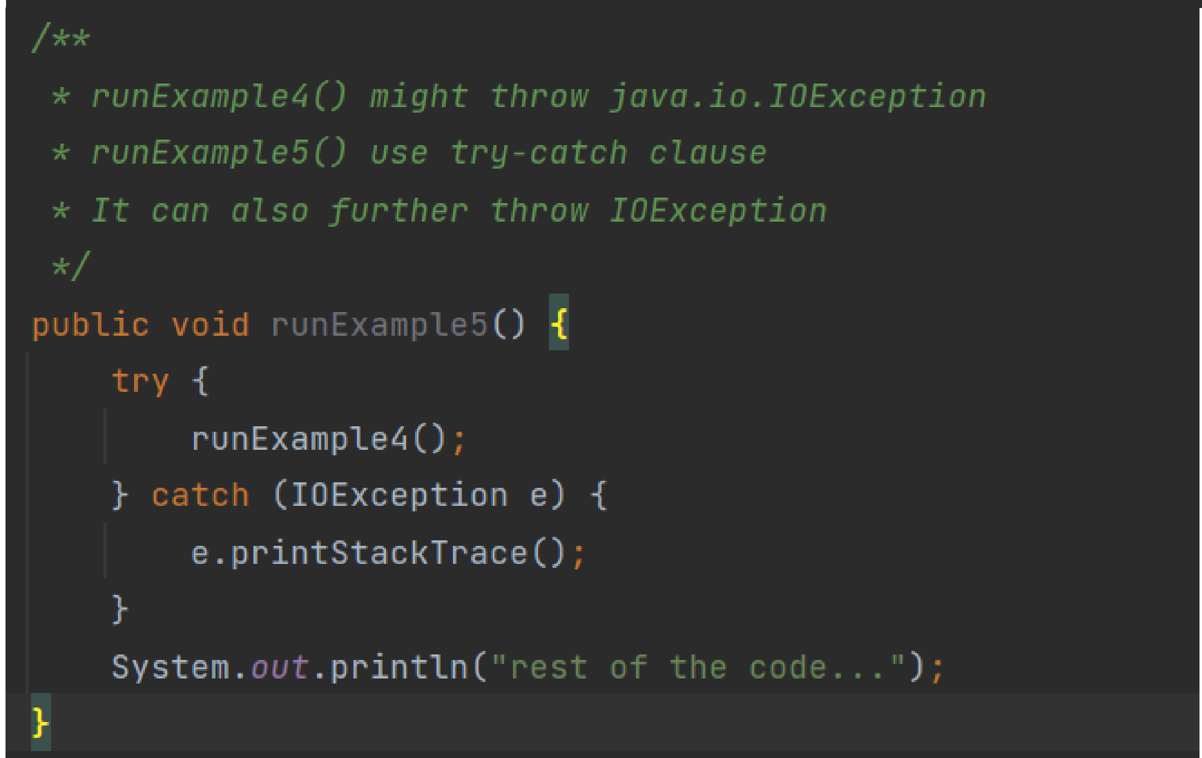
```java
/**
 * we explicitly throw an exception inside try block
 * and re-throw it in catch block
 */
public void runExample2() {
    try {
        throw new NullPointerException();
    }catch (NullPointerException e) {
        System.out.println(e);
        throw e;
    }
}
/**
 * validate() throw ArithmeticException
 * runExample3() does not handle it
 */
public void validate(String ssn) {
    if(ssn.length()!=11) {
        throw new ArithmeticException();
    }
}
public void runExample3() {
    validate( ssn: "123");
    System.out.println("rest of the code...");
}
```

Throws clause

- Throws keyword is used in signature of a method to declare an exception which might be thrown by the function
- We can declare multiple exceptions to be thrown by a function
- Syntax: The throws keyword is followed by class name of Exception to be thrown
- The throws keyword is used for checked exceptions

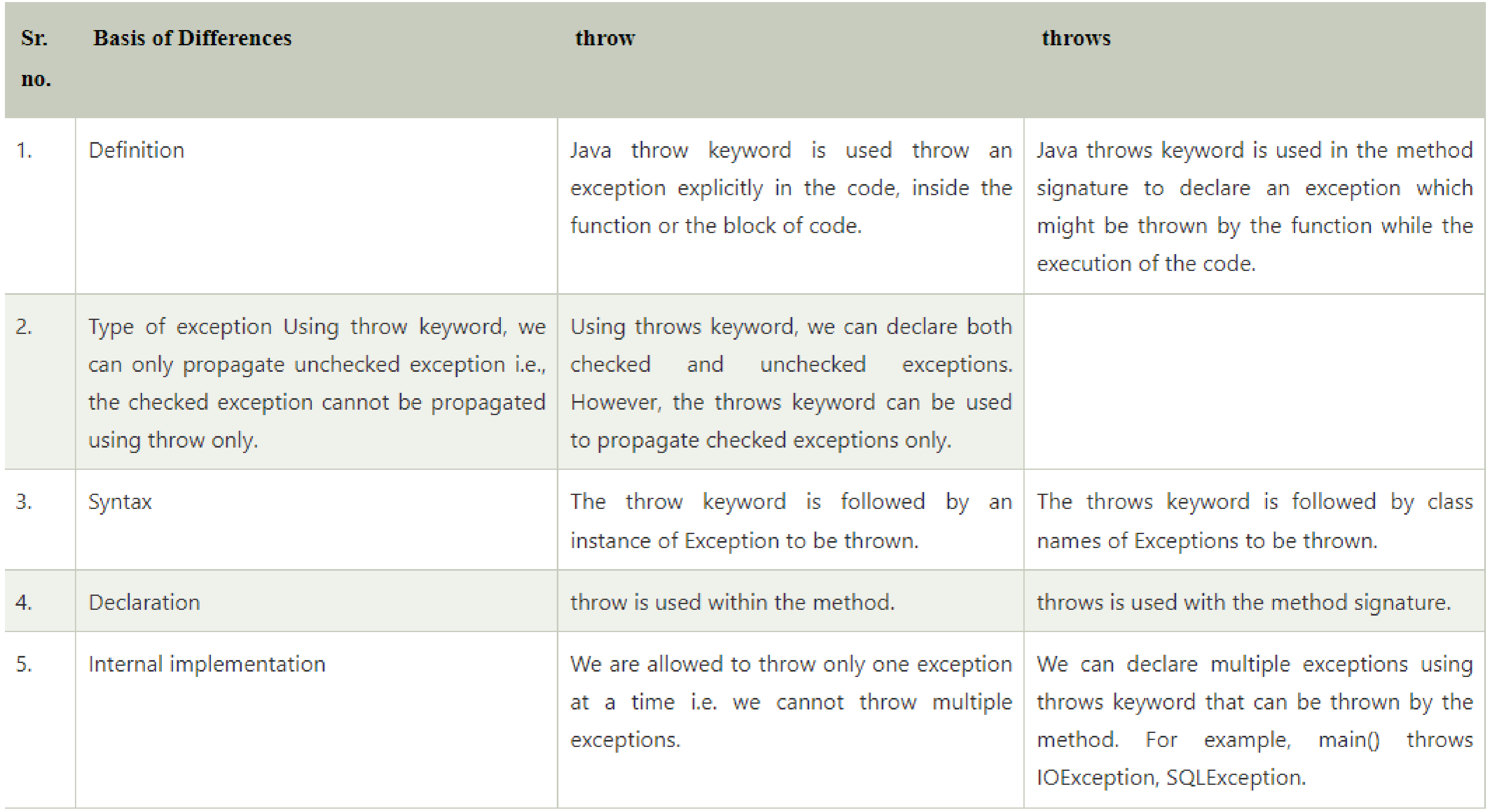


```java
public void runExample4() throws IOException {
    /**
     * You either add exception into method
     * signature or use try-catch clause
     */
    FileReader file = new FileReader( fileName: "C:\\Users\\abc.txt");
    BufferedReader fileInput = new BufferedReader(file);
    String line;
    while((line = fileInput.readLine())!=null){
        System.out.println(line);
    }
    fileInput.close();
}
```



```java
/**
 * runExample4() might throw java.io.IOException
 * runExample5() use try-catch clause
 * It can also further throw IOException
 */
public void runExample5() {
    try {
        runExample4();
    } catch (IOException e) {
        e.printStackTrace();
    }
    System.out.println("rest of the code...");
}
```

Throw vs Throws

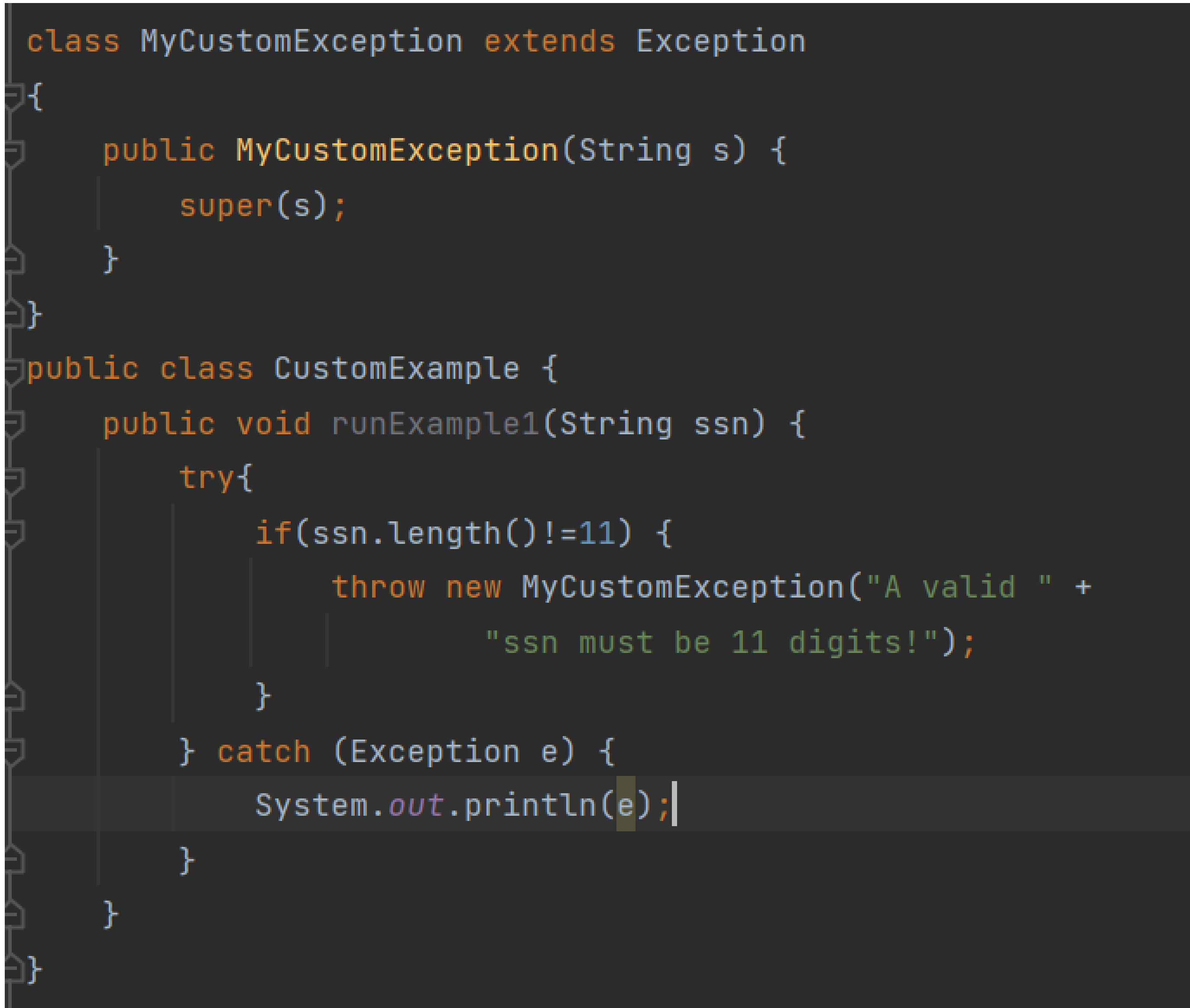


Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Src:https://www.javatpoint.com/difference-between-throw-and-throws-in-java

Custom Exception

- In Java, we can create our own exceptions.
- It needs to be derived from the Exception class.
- In which scenarios we want to create custom exceptions?
  - To catch and provide specific treatment to a subset of existing Java exceptions
  - When we define our own business logic



```java
class MyCustomException extends Exception
{
    public MyCustomException(String s) {
        super(s);
    }
}
public class CustomExample {
    public void runExample1(String ssn) {
        try{
            if(ssn.length()!=11) {
                throw new MyCustomException("A valid " +
                        "ssn must be 11 digits!");
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Exceptions - Summary

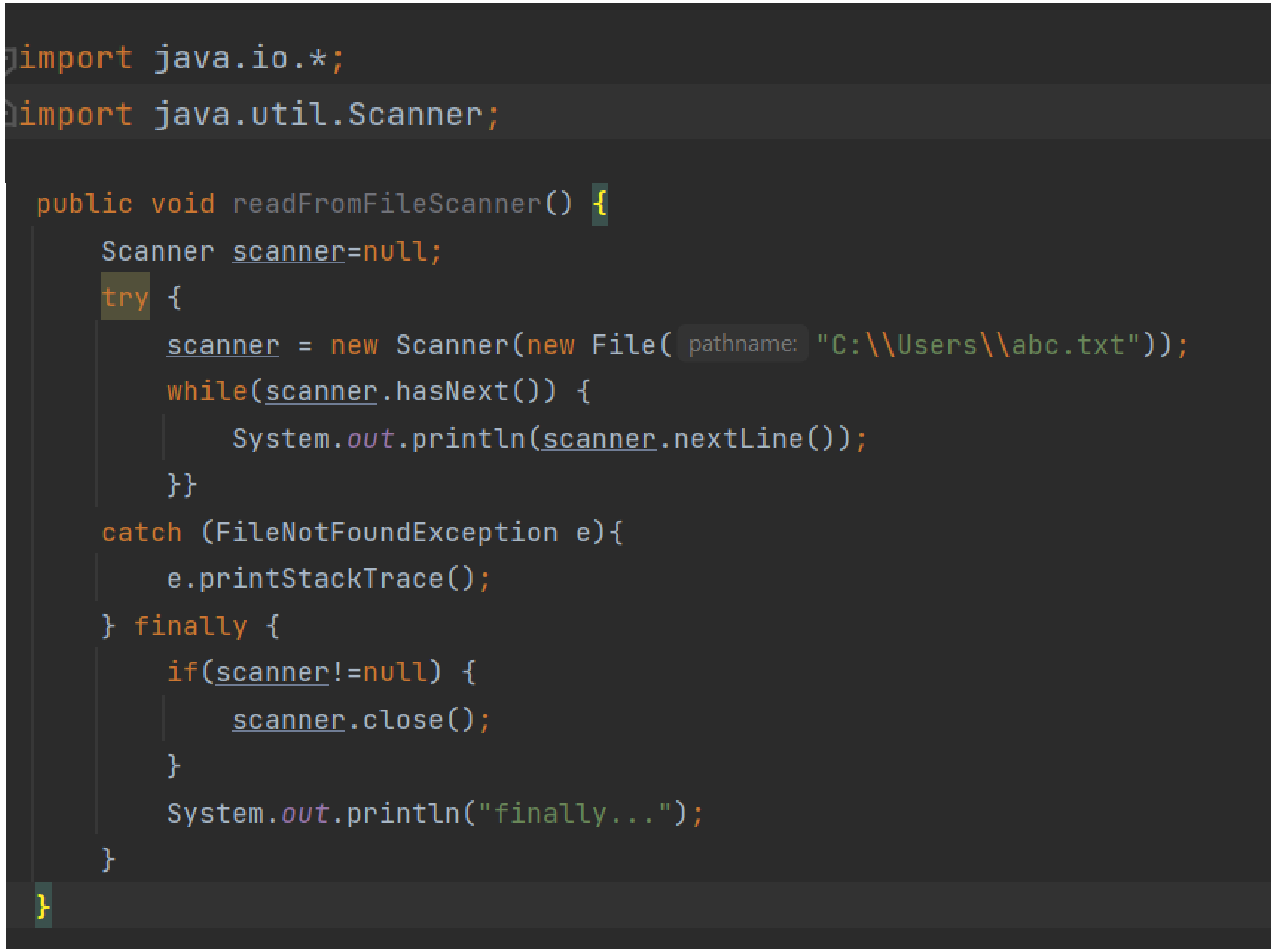
- When we call methods that throw checked exceptions, we must decide whether we want to throw the exception further, or process it
  - Try/Catch/Finally block to process it
  - Declare throws in signature of a method to throw it further
- Once exception handling is done, the exception object will be garbage collected.
- Exception handling is a large topic, and we will return to it later. For example:
  - Custom exceptions
  - Multi-catch
  - Try-with-resources
  - Chained exceptions

Read from File

- We already know Scanner!
  - You can create a Scanner from System.in
  - Here we learn how to create a Scanner from file.

- Try/catch/finally clause is used for exception handling

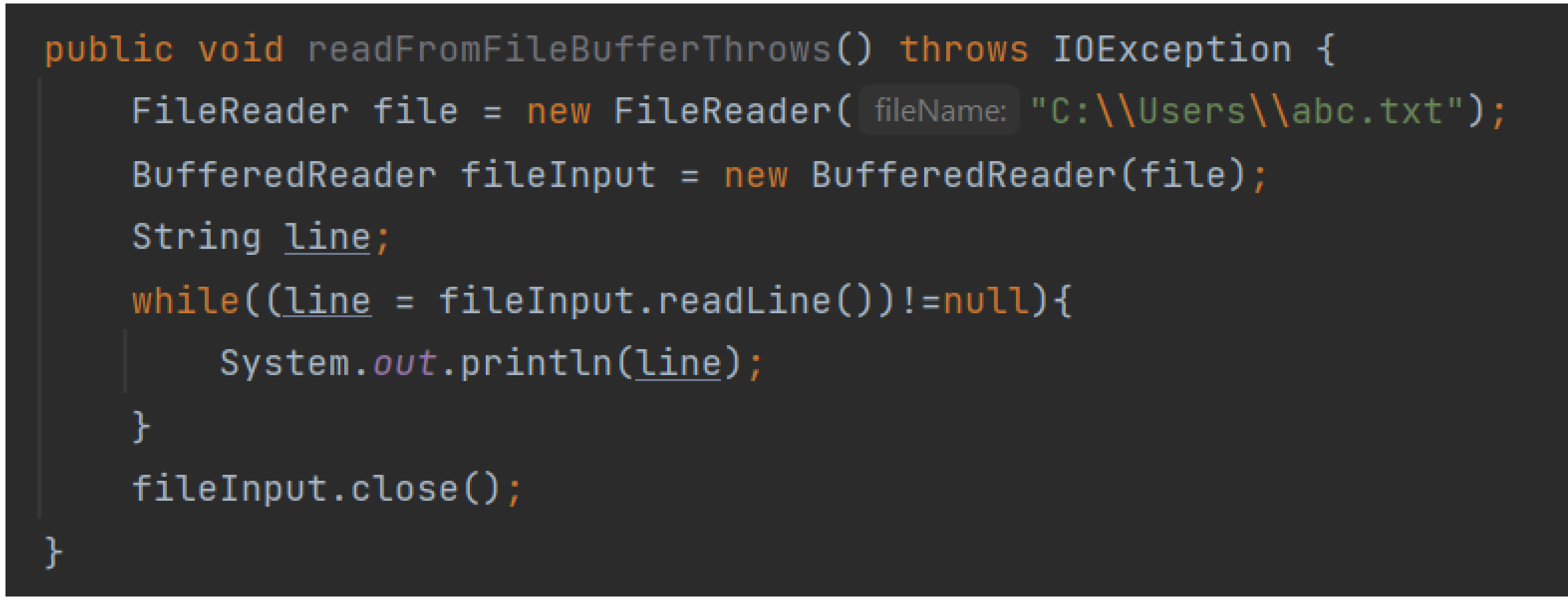
- FIleNotFoundException is checked exception



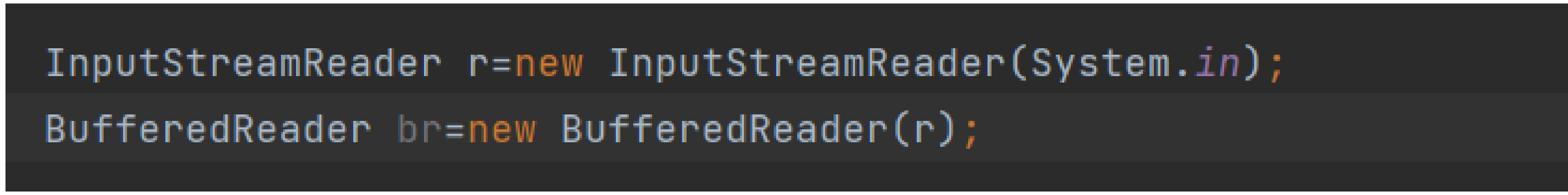
```
import java.io.*;
import java.util.Scanner;

public void readFromFileScanner() {
    Scanner scanner=null;
    try {
        scanner = new Scanner(new File( pathname: "C:\\Users\\abc.txt"));
        while(scanner.hasNext()) {
            System.out.println(scanner.nextLine());
        }}
    catch (FileNotFoundException e){
        e.printStackTrace();
    } finally {
        if(scanner!=null) {
            scanner.close();
        }
        System.out.println("finally...");
    }
}
```

Read from File

- We can use FileReader to read from file
  - FileReader is used to read data from a file.
  - BufferedReader class is used to read the text from a character-based input stream.
  - BufferedReader class can also be used to read from System.in



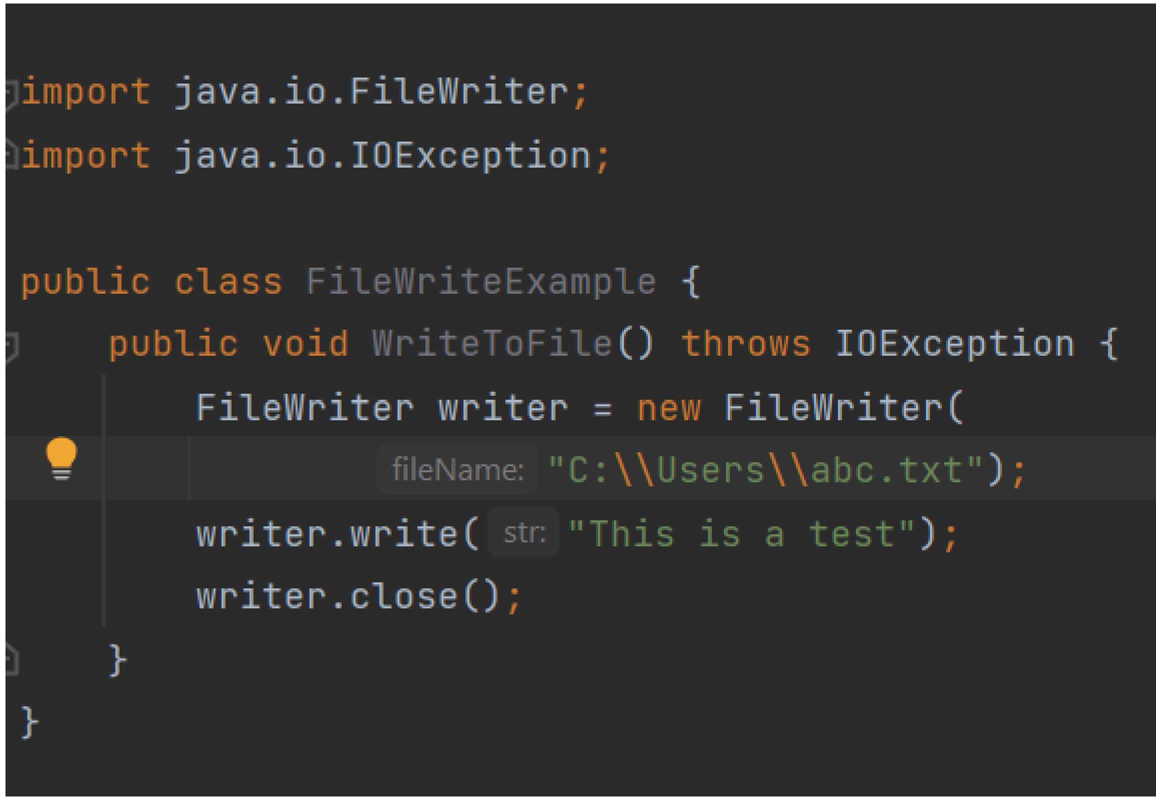
```java
public void readFromFileBufferThrows() throws IOException {
    FileReader file = new FileReader( fileName: "C:\\Users\\abc.txt");
    BufferedReader fileInput = new BufferedReader(file);
    String line;
    while((line = fileInput.readLine())!=null){
        System.out.println(line);
    }
    fileInput.close();
}
```



```java
InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);
```

Write to File

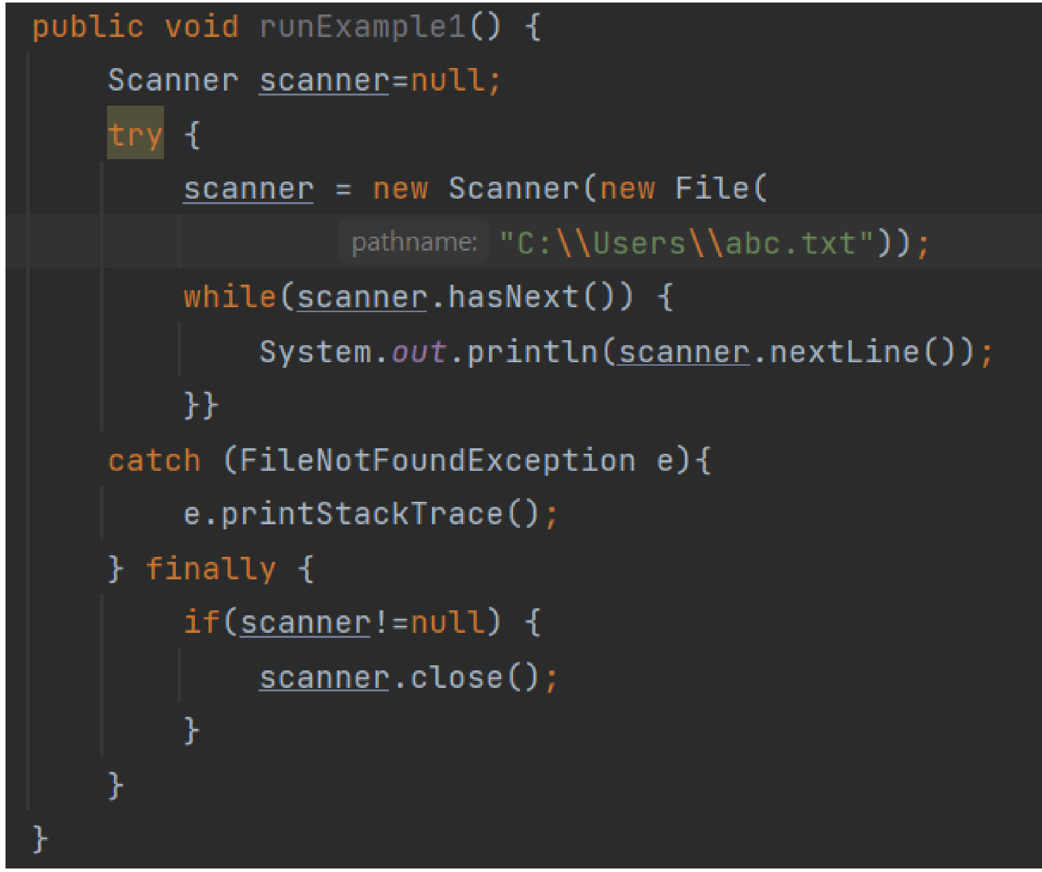
- FileWriter is used to write data to a file
- Note that IOException is further throwed



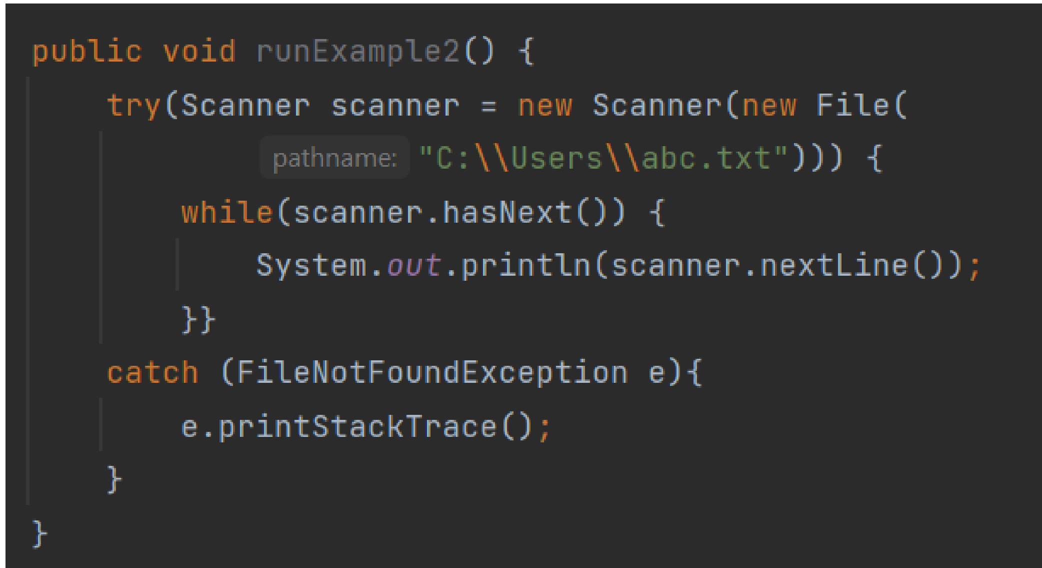
```java
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteExample {
    public void WriteToFile() throws IOException {
        FileWriter writer = new FileWriter(
                fileName: "C:\\Users\\abc.txt");
        writer.write( str: "This is a test");
        writer.close();
    }
}
```

Try-with-resources

- Replacing try-catch-finally with try-with-resources
- We can declare one or more resources
  - With try-catch-finally clause, you use finally block to ensure that a resource is closed
  - With try-with-resources, resource is closed automatically



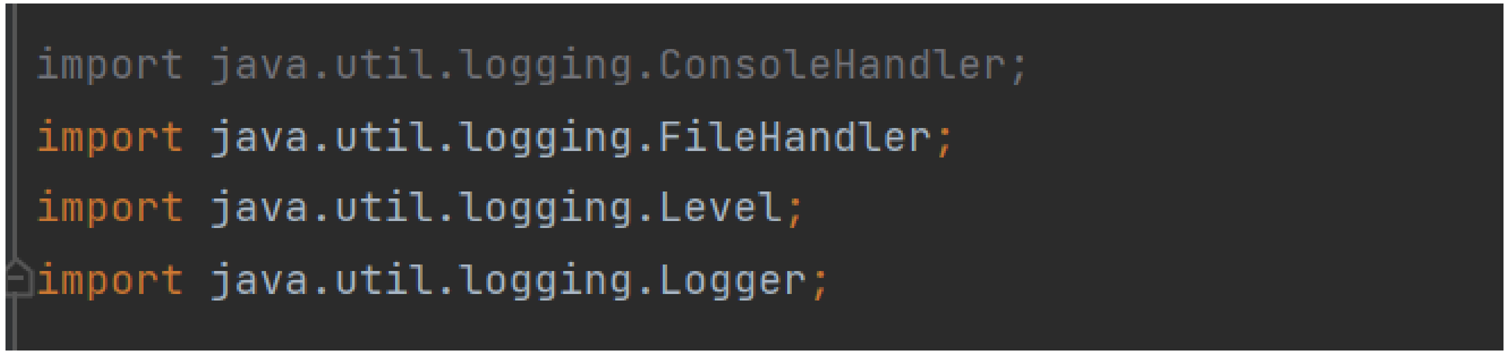
```
public void runExample1() {
    Scanner scanner=null;
    try {
        scanner = new Scanner(new File(
                pathname: "C:\\Users\\abc.txt"));
        while(scanner.hasNext()) {
            System.out.println(scanner.nextLine());
        }}
    catch (FileNotFoundException e){
        e.printStackTrace();
    } finally {
        if(scanner!=null) {
            scanner.close();
        }
    }
}
```



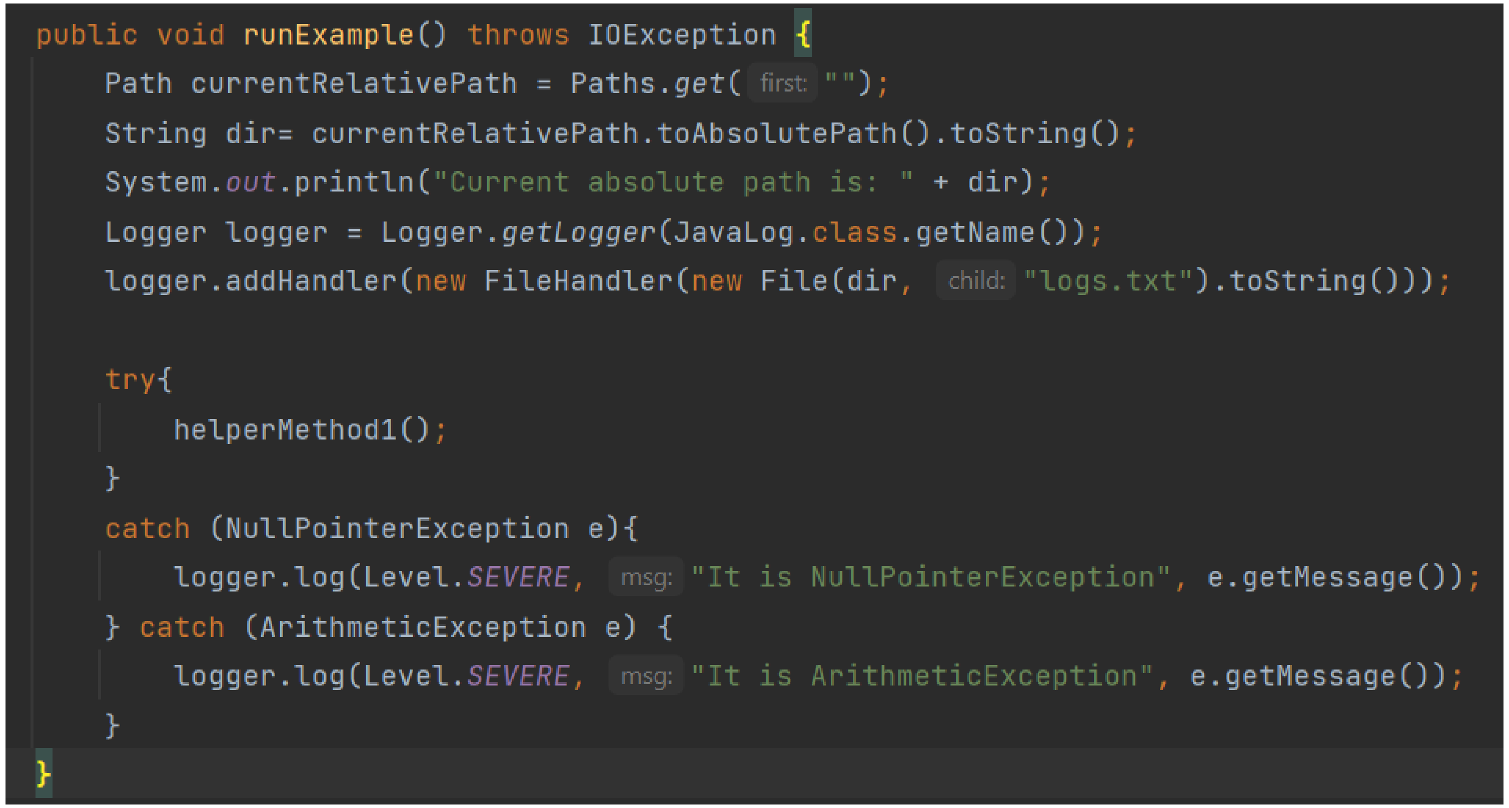
```
public void runExample2() {
    try(Scanner scanner = new Scanner(new File(
            pathname: "C:\\Users\\abc.txt"))) {
        while(scanner.hasNext()) {
            System.out.println(scanner.nextLine());
        }}
    catch (FileNotFoundException e){
        e.printStackTrace();
    }
}
```

Java Logger

- The need for Log capture
  - Recording unusual circumstances or errors
  - Getting the infor about what is going in the application
- Log level
  - SEVERE(highest value)
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (lowest value)
- Java provides Logger class in package java.util.logging.Level
- Apache Log4j is popular API which we will introduce later together with Maven



```java
import java.util.logging.ConsoleHandler;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
```

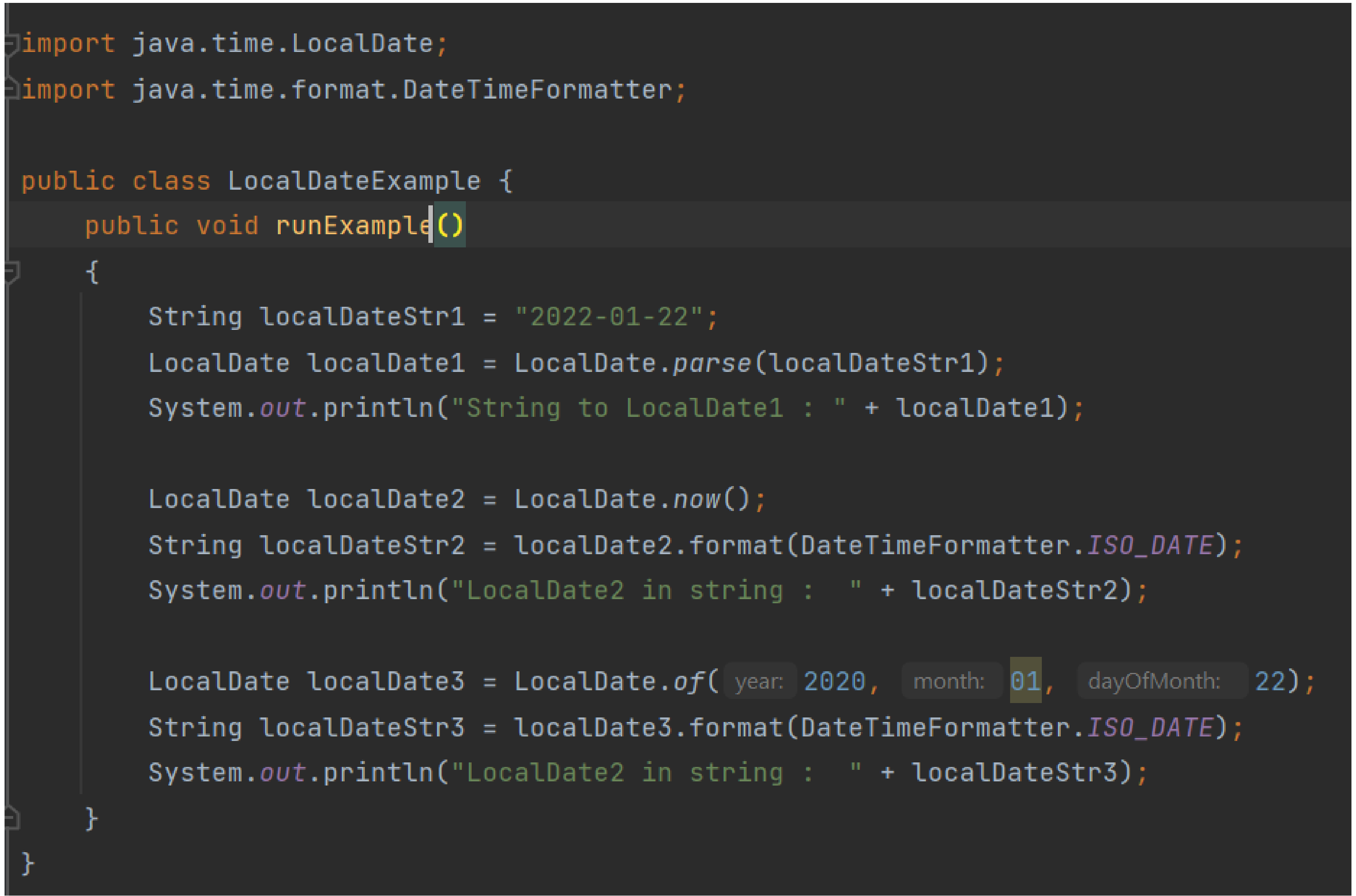


```java
public void runExample() throws IOException {
    Path currentRelativePath = Paths.get( first: "");
    String dir= currentRelativePath.toAbsolutePath().toString();
    System.out.println("Current absolute path is: " + dir);
    Logger logger = Logger.getLogger(JavaLog.class.getName());
    logger.addHandler(new FileHandler(new File(dir,  child: "logs.txt").toString()));

    try{
        helperMethod1();
    }
    catch (NullPointerException e){
        logger.log(Level.SEVERE,  msg: "It is NullPointerException", e.getMessage());
    } catch (ArithmeticException e) {
        logger.log(Level.SEVERE,  msg: "It is ArithmeticException", e.getMessage());
    }
}
```

LocalDate

- Java struggled for a long time to sort out data that had to deal with time and place.
- The java.time package contains a lot of goodies you can use
- In this week’s assignments, the LocalDate class in particupar is relevant. It represents a date
- Java LocalDate class is an immutable class that represents Date with a default format of yyyy-mm-dd

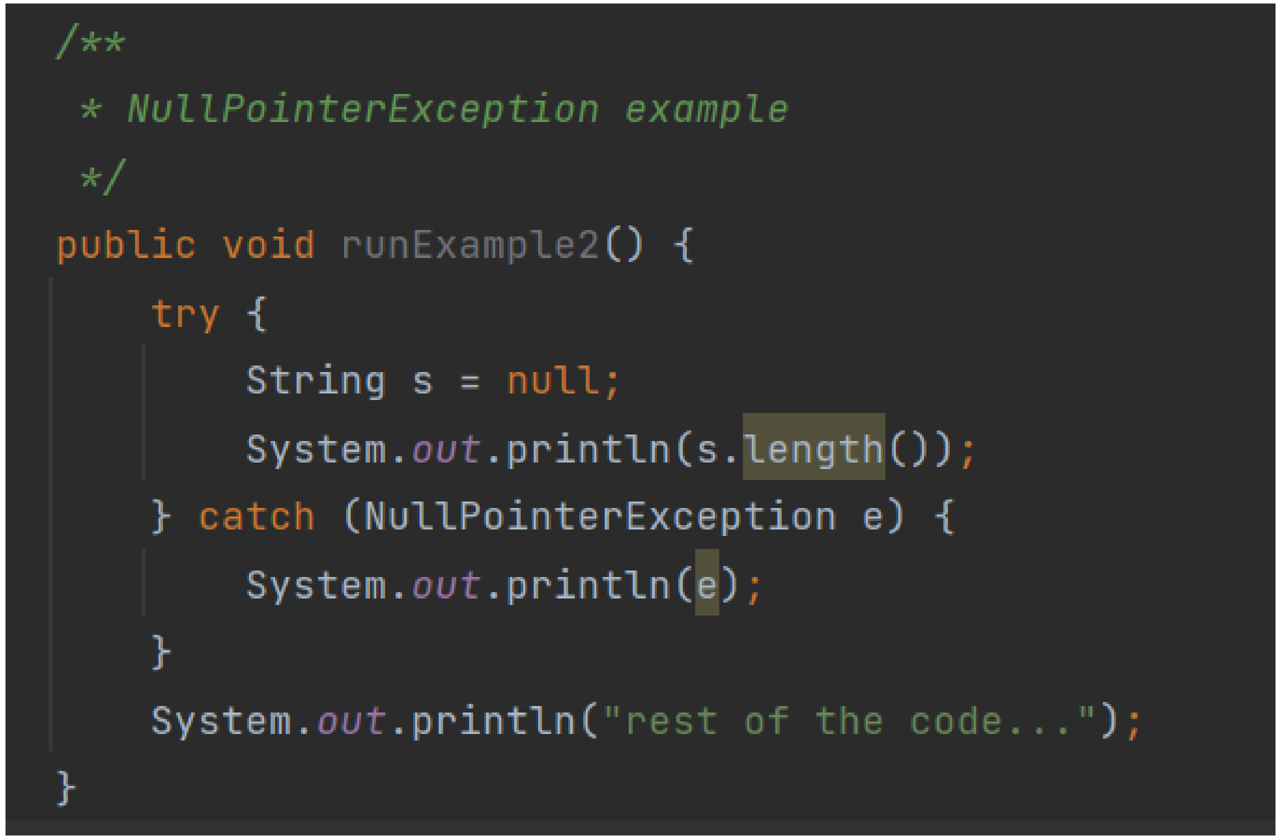
Create a LocalDate object from textual representation



```java
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class LocalDateExample {
    public void runExample()
    {
        String localDateStr1 = "2022-01-22";
        LocalDate localDate1 = LocalDate.parse(localDateStr1);
        System.out.println("String to LocalDate1 : " + localDate1);

        LocalDate localDate2 = LocalDate.now();
        String localDateStr2 = localDate2.format(DateTimeFormatter.ISO_DATE);
        System.out.println("LocalDate2 in string :  " + localDateStr2);

        LocalDate localDate3 = LocalDate.of( year: 2020,  month: 01,  dayOfMonth: 22);
        String localDateStr3 = localDate3.format(DateTimeFormatter.ISO_DATE);
        System.out.println("LocalDate2 in string :  " + localDateStr3);
    }
}
```

Null

- You probably (should?) remember the null from DB topic
- We have the same phenomenon in Java (and JavaScript)
- NullPointerException is a very common exception to encounter
- Occurs when you perform something with an object, for example calls a method defined in the objectm but the object is null



```
/**
 * NullPointerException example
 */
public void runExample2() {
    try {
        String s = null;
        System.out.println(s.length());
    } catch (NullPointerException e) {
        System.out.println(e);
    }
    System.out.println("rest of the code...");
}
```

Before we end

- Goals for this session:
  - I understand Java Exceptions
  - I know how to use Exception Handling
  - I know how to read from and write to file
  - I know how to log using Java logging class
  - I know LocalDate class
  - I know Null

Remember, do not just read code, play with it.
Good luck with the tasks!

Remember, there is help available all week, use Mattermost or GitHub.