

Exceptions & File I/O: Reading Files

Module 1: 16

Week 4 Overview

Monday

Exceptions
&
File IO
Reading Files

Tuesday

File IO Writing
Files

Wednesday

Review

Thursday

M1 Capstone

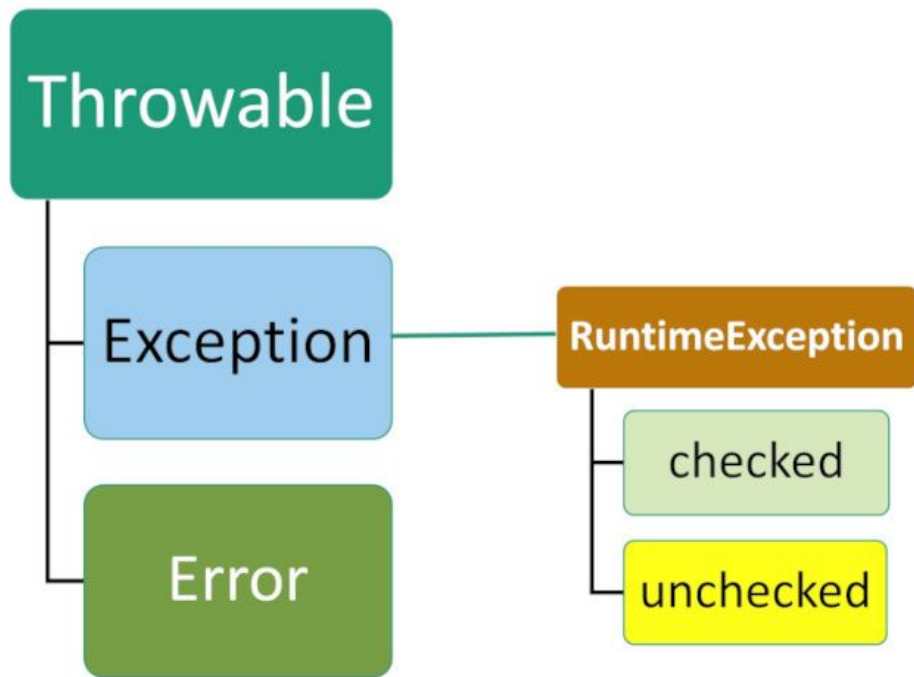
Friday

M1 Capstone

Today's Objectives

1. Exception Handling
2. File I/O - Reading Files

Java Exception Handling



try
catch
finally
throw
throws

Types of Errors

Run Time Errors

Occurs while the program is being executed by the JVM.

Caused by the JVM being asked to perform an operation that is not possible.

Divide by Zero
Null Pointer

Compile Time Errors

Occurs when javac tries to compile the source code to byte code.

Caused by not following the correct syntax in source code.

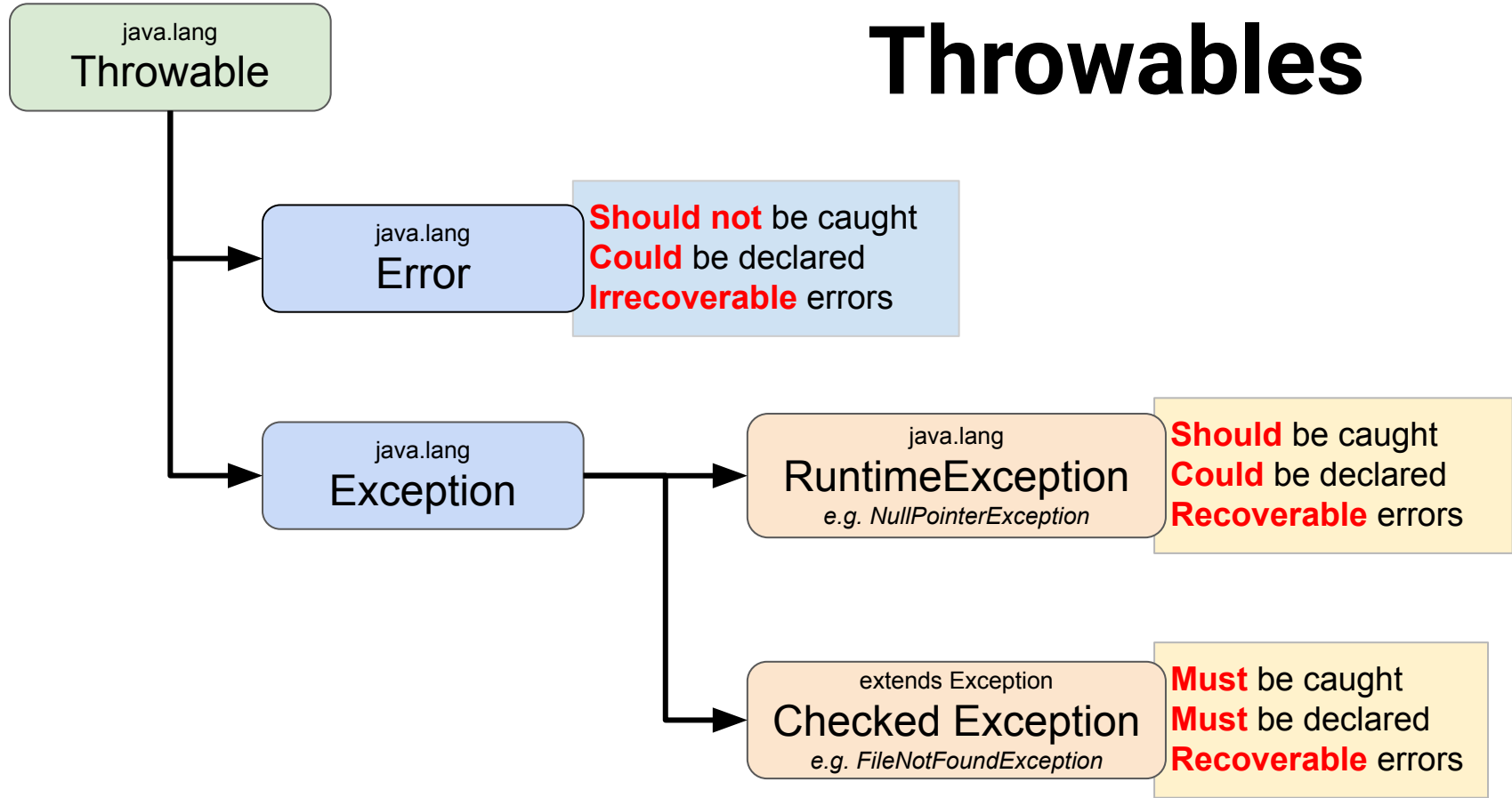
Syntax Errors
Semantic Errors

Error and Exception

All Exceptions and Errors in Java are subclasses of the class *Throwable*.

Exception	Error
An unexpected situations that occur while a program is executing. It is what happens when something is unexpected or goes wrong, such as the index of an array being out of bounds.	Used by the JVM to indicate errors that are associated with the runtime environment, such as running out of memory or other resources.
Possible to recover	Impossible to recover
Can be caught and handled	Should not be handled
Occur at compile or runtime	Occur at runtime
Caused by code or data	Caused by the running environment

Throwables

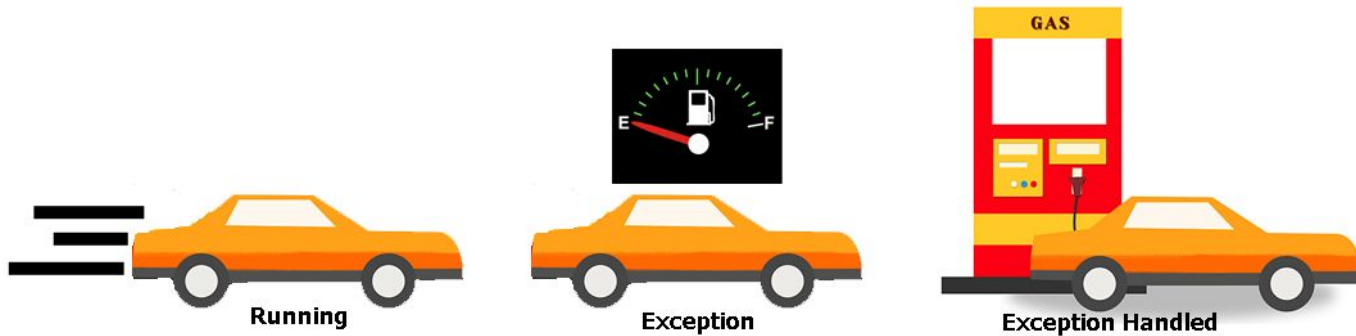


Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions.

An exception is represented by an object of type Exception that contains information about the error.

Exceptions can be *caught* and *handled* to allow the program to continue running.



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:
Add throws declaration
Add catch clause to surrounding try
Press F2 for focus

- 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)
```

Runtime Exception

Superclass: `java.lang.RuntimeException`

`java.lang`

RuntimeException

e.g. NullPointerException

Should be caught
Could be declared
Recoverable errors

Runtime Exceptions (*or unchecked exceptions*) can be *thrown* from any method, do not need to be declared, and do not have to *caught* with a try...catch. If a runtime exception is not caught it will *throw* to the JVM and the application will stop (crash).

Common Runtime Exceptions

- `ArrayIndexOutOfBoundsException`
- `NullPointerException`
- `ClassCastException`
- `NumberFormatException`
- `NoSuchElementException`

Checked Exception

Superclass: `java.lang.Exception`

extends `Exception`

Checked Exception

e.g. `FileNotFoundException`

Must be caught
Must be declared
Recoverable errors

Checked Exceptions are *thrown* from methods that *declare* them. They **must be handled** by either *catching* them using a `try...catch` or by *declaring* it as throwable from the method.

Common Checked Exceptions

- `ClassNotFoundException`
- `FileNotFoundException`
- `SQLException`
- `IOException`

Exception Handling

Exception Handling is dealing with unexpected problems in an application so the program does not crash.

If exceptions are not handled, then the application will terminate (crash).

When an unexpected event happens in Java an Exception is ***Thrown***.

The *thrown* exception includes an *Exception Object* that contains details about what happened and a ***Stack Trace*** that details where it occurred in the code.

Thrown Exception can be ***caught***, the exception object can be used to determine what happened, and then steps taken to deal with the error.

Exceptions are caught and dealt with using a ***Try...Catch*** block.

Exception Terminology

Throw

A keyword used to create an exception

Declared

When a method includes the throws keyword in its method signature to indicate it may create an exception.

Stack Trace

Details of what methods and lines of code were being run when the exception occurred.

Thrown

An exception has been created and needs to be handled.

Caught

When code exists to handle the exception to allow execution to continue.

Handled

When code has caught an exception and taken some action to deal with it and execution of the program has continued.

Throws

A keyword used in a method signature to indicate a method may create an exception



Try...Catch Block

Risky code is surrounded with a try...catch. The **try** identifies a block of code that may cause an exception, and the **catch** block identifies a block of code to run if an exception occurs.

```
try {  
    Scanner in;  
    String choice = in.nextLine();  
    int x = Integer.parseInt(choice);  
} catch (NullPointerException e) {  
    Code to handle the exception  
}
```

NullPointerException

Code to handle the exception



When an exception occurs in the try, all following lines of code are skipped and the catch is immediately executed.

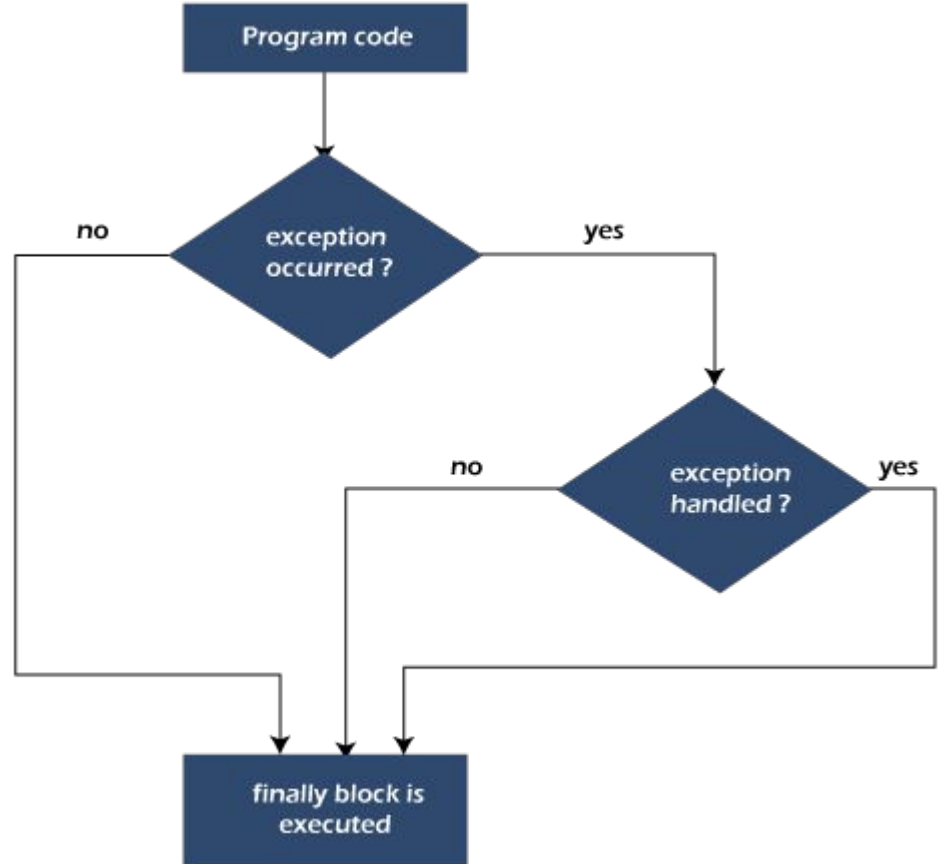
[Visual Explanation](#)

Try...Catch...Finally

```
try {  
    risky code  
} catch (NullPointerException e) {  
    code to handle a NullPointerException  
} finally {  
    code in finally will always be executed  
}
```

Code in the finally block ALWAYS runs, even if the exception is unhandled and crashes the program.

The only time code in a finally will not execute is if the program is exited or killed (power outage, ended from task manager, etc.) while the try block is executing.



Parts of a Try...Catch...Finally

```
try {  
    risky code  
} catch (NullPointerException e) {  
    code to handle a NullPointerException  
} catch (FileNotFoundException e) {  
    code to handle a FileNotFoundException  
} catch (Exception e) {  
    code to handle any other Exception  
} finally {  
    code in finally will always be executed  
}
```

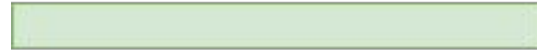
The try block identifies a block of code that may throw an exception that should be handled.

Multiple catch statements can be chained to handle different exceptions from the same try block. The first matching catch will be executed, so multiple catch statements must be organized in least to most specific.

The optional finally block identifies code that will always be run whether or not an exception is thrown.



try {



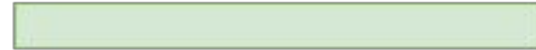
} catch (Exception e) {



}



try {



} catch (Exception e) {



} finally {



}



- * no exceptions
- * an exception occurs in the blue block
- * an exception occurs in the blue block and the yellow block contains a `return` statement

Throw vs Throws

throw: A keyword used *in a method* to create an exception.

```
throw new MyException();
```

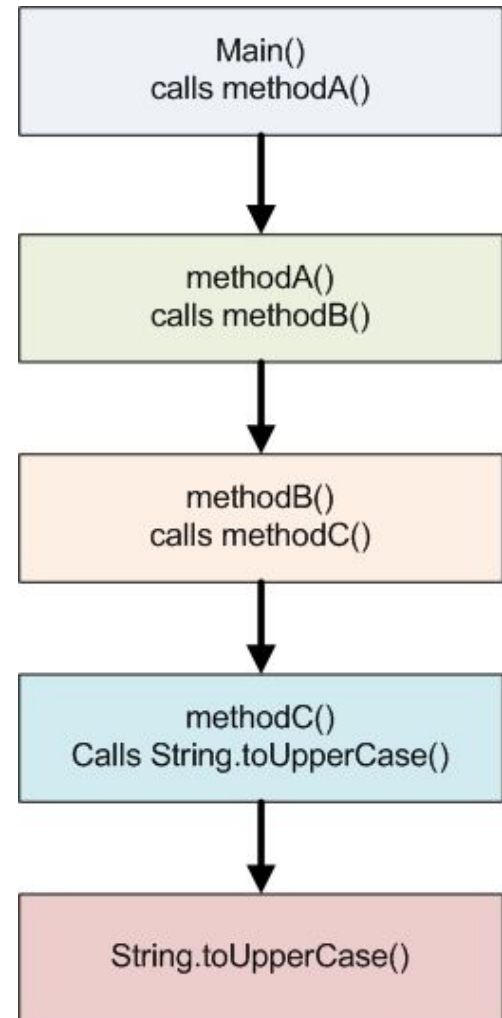
throws: A keyword used *in a method signature* to declare the method may throw an exception.

```
public void myMethod() throws MyException
```

Throw	Throws
Keyword used to explicitly throw an exception	Keyword used to declare an exception
Cannot propagate Checked Exceptions on its own	Can propagate Checked Exceptions
Followed by an instance	Followed by a class
Used as a statement within a method	Used in the method signature
Cannot throw multiple exceptions	Can be used to declare multiple exceptions

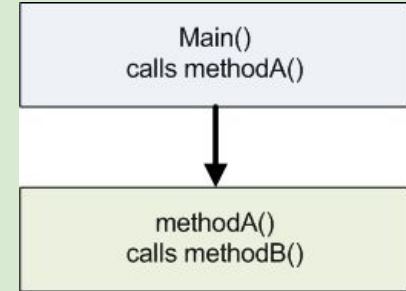
Call Stack

Methods call other methods. As each method is called it is added to the *Call Stack*, which is a map of what code is currently executing and the path the code took to get there.



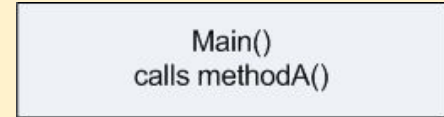
When a method is called from another it is added to the stack.

```
main() {  
    methodA()  
}
```



When a method returns it is removed from the stack.

```
methodA() {  
    return;  
}
```



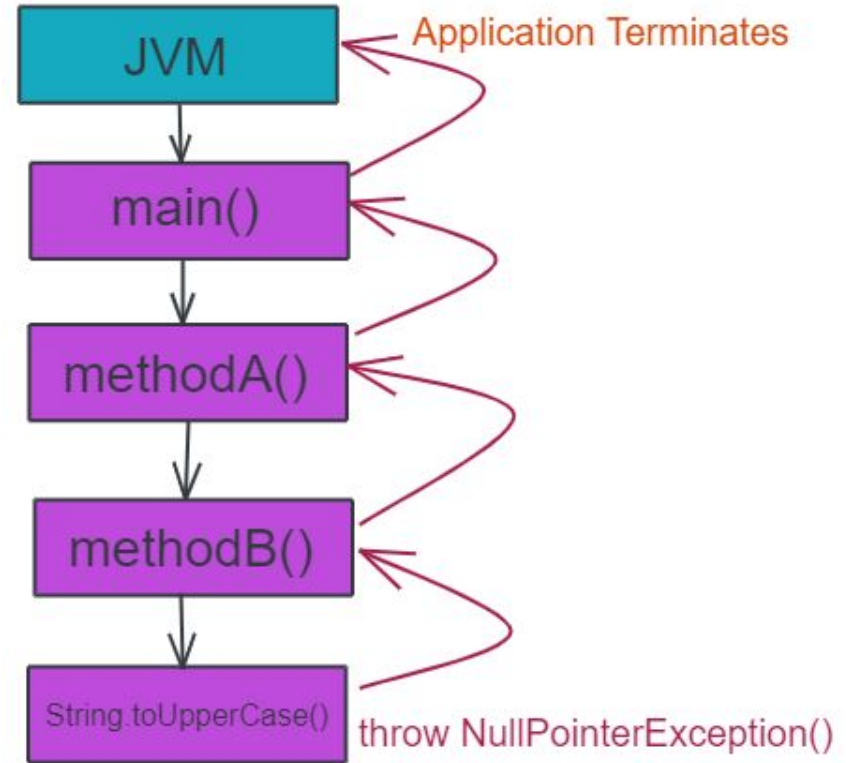
If methods are not allowed to return (note void methods return automatically when the method ends) then the call stack continues to grow. When it grows to large it results in a **StackOverflow Error** that cannot be recovered and the application will terminate in a crash.

Exception Propagation

When an exception is thrown, and is not handled, it propagates up the call stack.

The exception can be caught by a try...catch in any method in call stack and handled, which stops the propagation.

If the exception is not handled and allowed to propagate the JVM, then the application terminates in a crash.



Stack Trace

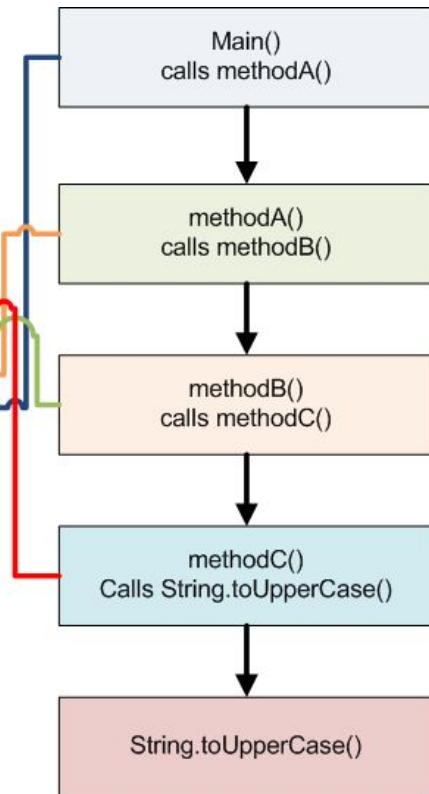
NullPointerException Was Thrown

The Exception was thrown in methodC()

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint
    at com.techelevator.exceptions.ExceptionStackExamples.methodC(ExceptionStackExamples.java:33)
    at com.techelevator.exceptions.ExceptionStackExamples.methodB(ExceptionStackExamples.java:23)
    at com.techelevator.exceptions.ExceptionStackExamples.methodA(ExceptionStackExamples.java:19)
    at com.techelevator.exceptions.ExceptionStackExamples.main(ExceptionStackExamples.java:10)
```

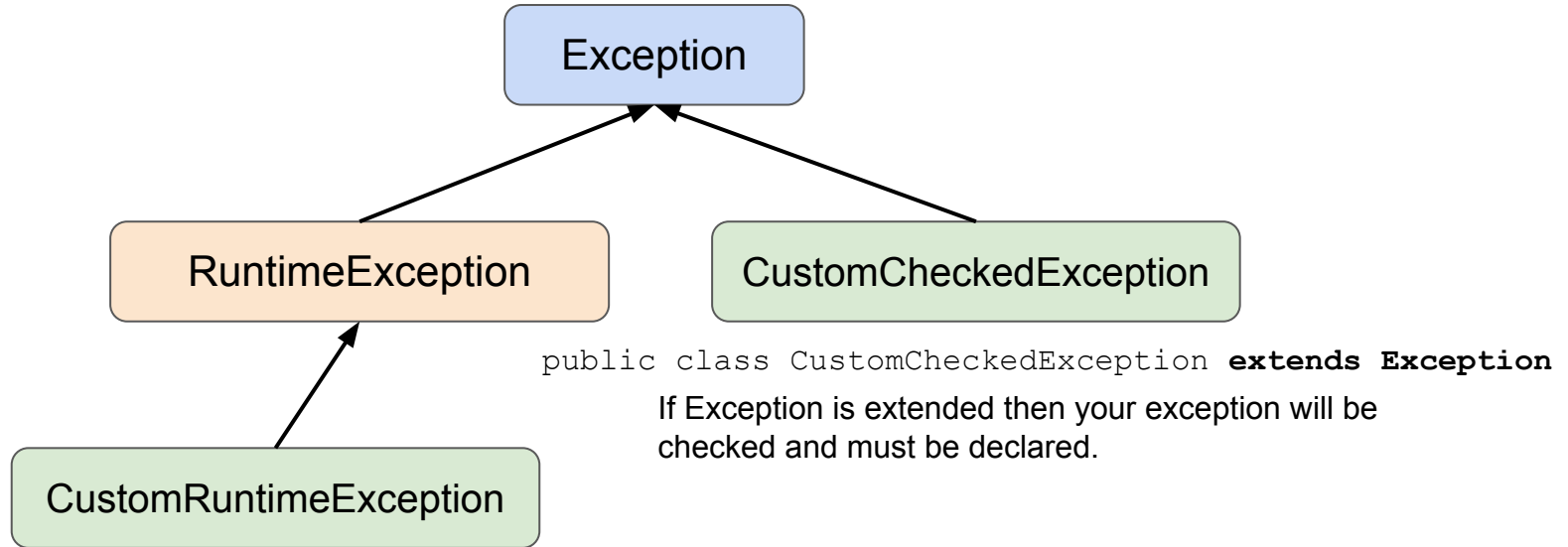
A stack trace is information included in an exception that shows the exception type thrown and call stack that existed when the exception occurred. The call stack is read bottom to top.

It's not always this simple. Frameworks and libraries can make call stacks long and often show methods above yours in the stack, but you will always find your method that caused the error somewhere in the stack and start reading from that point.



Custom Exceptions

Custom exceptions can be created by extending either `Exception` or `RuntimeException`. Custom exceptions are used to communicate exceptions in your application that are specific to it.



```
public class CustomCheckedException extends Exception
```

If `Exception` is extended then your exception will be checked and must be declared.

```
public class CustomRuntimeRuntimeException extends RuntimeException
```

If `RuntimeException` is extended then your exception will be unchecked and does not need to be declared.

Possible Interview Questions:

- **What is the difference between exception and error in Java?**
- **Why do we need exception handling in Java?**
- **Name the different types of exceptions in Java**
- **Describe the use of the throw keyword.**
- **What is the difference between the throw and throws keyword in Java?**

Possible Interview Questions:

- **What is the difference between exception and error in Java?**

- Exception- An unexpected situations that occur while a program is executing. It is what happens when something is unexpected or goes wrong, such as the index of an array being out of bounds.
- Error- Used by the JVM to indicate errors that are associated with the runtime environment, such as running out of memory or other resources.

- **Why do we need exception handling in Java?**

- If there is no try and catch block while an exception occurs, the program will terminate. Exception handling ensures the smooth running of a program without program termination.

- **Name the different types of exceptions in Java**

- Checked: Occur during the compilation. Here, the compiler checks whether the exception is handled and throws an error accordingly.
- Unchecked: Occur during program execution. These are not detectable during the compilation process.

- **What is the difference between the throw and throws keyword in Java?**

- throws keyword is used with method signature to declare the exceptions that the method might throw whereas throw keyword is used to disrupt the flow of the program and handing over the exception object to runtime to handle it.

File 10

Reading Files



So far we have been able to get input from the user through `Scanner(System.in)` and `System.out`

The `System` class (`java.lang.System`) is a class that provides methods:

- `out` (`PrintStream` object)
- `err` (`PrintStream` object)
- `in` (`InputStream` object)

The `PrintStream` class (`java.lang.PrintStream`) is a class that provides methods:

- `print()`
- `println()`

The `InputStream` class (`java.lang.InputStream`) is a class that provides methods:

- `read()`
- `close()`
- `skip()`

`Scanner input = new Scanner(System.in);`
creates a new `Scanner` instance that reads from the standard input stream of the program. (aka data from keystrokes)

Java.io Library (<= link to java docs)

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.

We will focus on:

Today

java.io.File (An abstract representation of file and directory pathnames.)

java.io.PrintWriter (Prints formatted representations of objects to a text-output stream.)

Tomorrow

A file is an ordered and named collection of sequential bytes that has persistent storage.

3 basic file operations:

Read

Write

Seek

Methods exist to read all text in quickly with one line of code and dump it all into memory. (Yikes! What if it is a large file??) This would be like sitting to watch a Netflix movie and waiting for the entire movie to load before you start watching it.

File I/O

java.io.File

.exists()

.isFile()

File myFile = new File(pathToFile);

Note - File objects can only tell you information about the file. To open it, pass it to a Scanner object.

To avoid dumping to the heap, our new File object can be passed to Scanner as an input stream.

```
try (Scanner fileScanner = new Scanner(myFile))  
{  
  
    while (fileScanner.hasNextLine()) {  
        String line = fileScanner.nextLine();  
    } catch (FileNotFoundException ex) {  
  
    }
```

Streams have an end-of-file marker or end-of-stream marker to indicate when the program reaches the end of the stream.

Some objects Java implicitly cleanup any memory that they are utilizing while others require explicit cleanup.

When unused objects are no longer needed the memory occupied needs to be reclaimed. The JVM automatically releases memory that sits on the heap through a process called Garbage Collection.

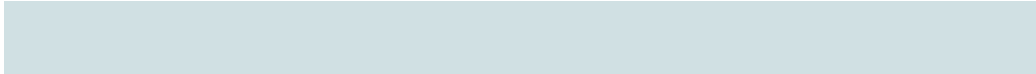
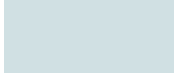
Other objects, such as files and connections, **require an explicit** release of resources. They need to be Disposed.

Original:

```
Scanner fileScanner = new Scanner(inputFile);  
fileScanner.close();
```

Using try-with-resource:

Java includes the AutoClosable interface to allow some objects to be closed for us automatically when we use the **try-with-resources structure**

```
try(Scanner fileScanner = new Scanner(inputFile)) {  
      
     }  
}
```

Handling exceptions when reading from a file stream

Exceptions can often occur when reading streams.

1. Directory not found
2. End of stream reached
3. File not found
4. Path too long (windows only)

Step 1: get the filename and path as a string

```
System.out.println("What is the file path?");  
Scanner input = new Scanner(System.in);  
String path = input;
```

Step 2: create a file object and pass it the filename

```
File file = new File(path);
```

Step 3: Open the file with a scanner in a try-with-resource

```
try(Scanner fileScanner = new Scanner(file){
```

Step 4: Loop while hasNextLine() is true

```
while(fileScanner.hasNextLine()){
```

Step 5: use nextLine() to read the next line from the file

```
String lineFromFile = fileScanner.nextLine();  
}  
} catch(FileNotFoundException e){  
    System.out.println("File not found");  
}
```

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, rtn.txt is located in the project root, we can refer to it like so:

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

Name

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

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

```
File inputFile = new  
File("resources/rtn.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.