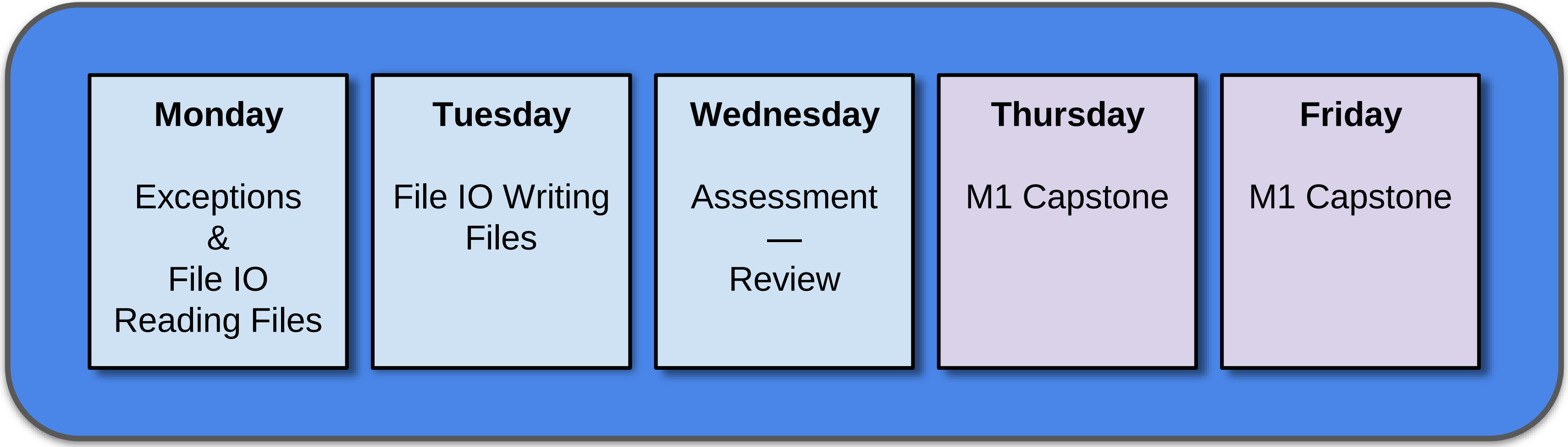
Exceptions &
File I/O:Reading Files

Module 1: 16

Week 4 Overview



Monday	Tuesday	Wednesday	Thursday	Friday
Exceptions & File IO Reading Files	File IO Writing Files	Assessment — Review	M1 Capstone	M1 Capstone

Today’s Objectives

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

Exceptions - 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.

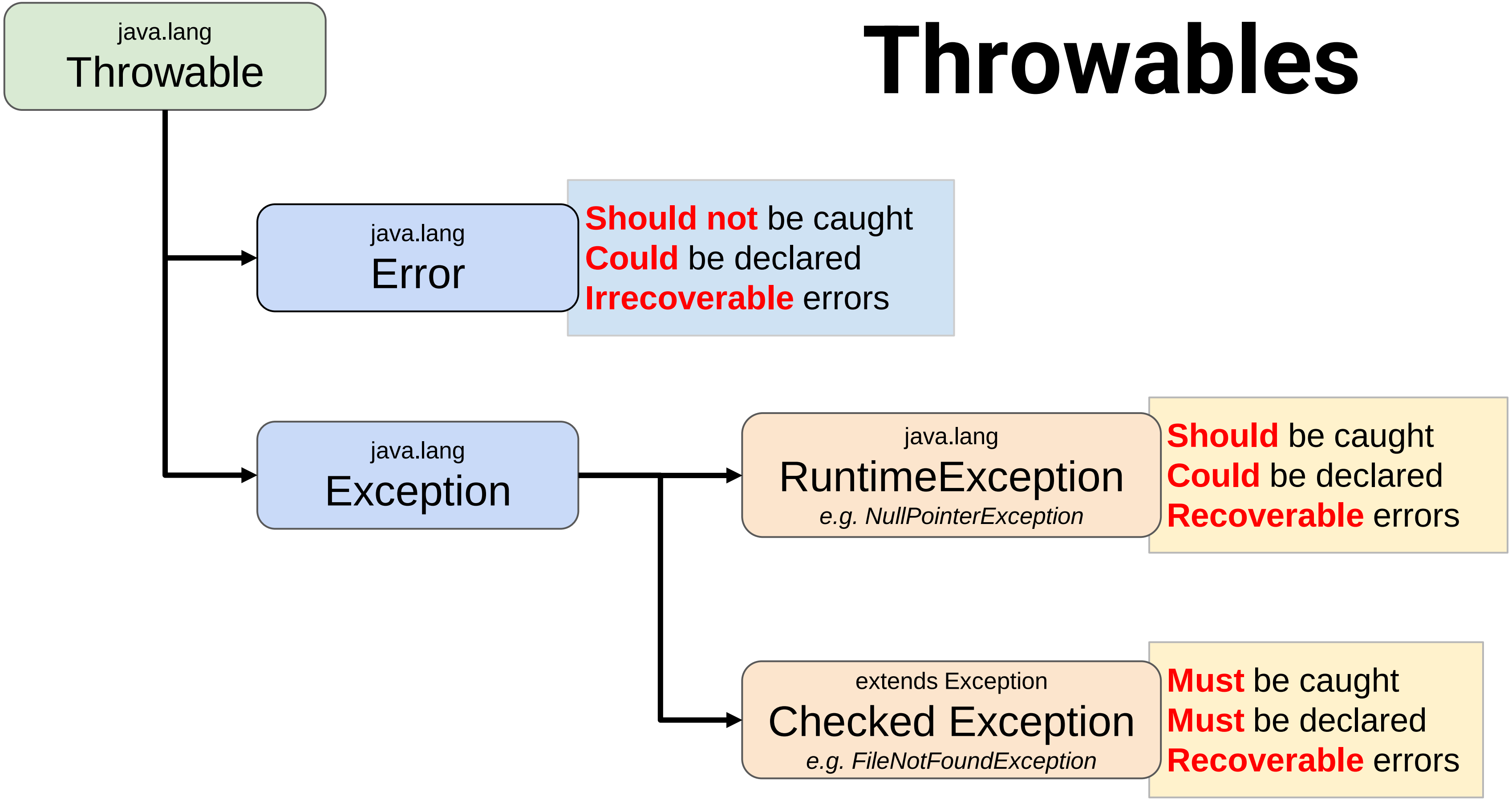
Syntax Errors
Semantic Errors

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



java.lang
Throwable

java.lang
Error

Should not be caught
Could be declared
Irrecoverable errors

java.lang
Exception

java.lang
RuntimeException
e.g. NullPointerException

Should be caught
Could be declared
Recoverable errors

extends Exception
Checked Exception
e.g. FileNotFoundException

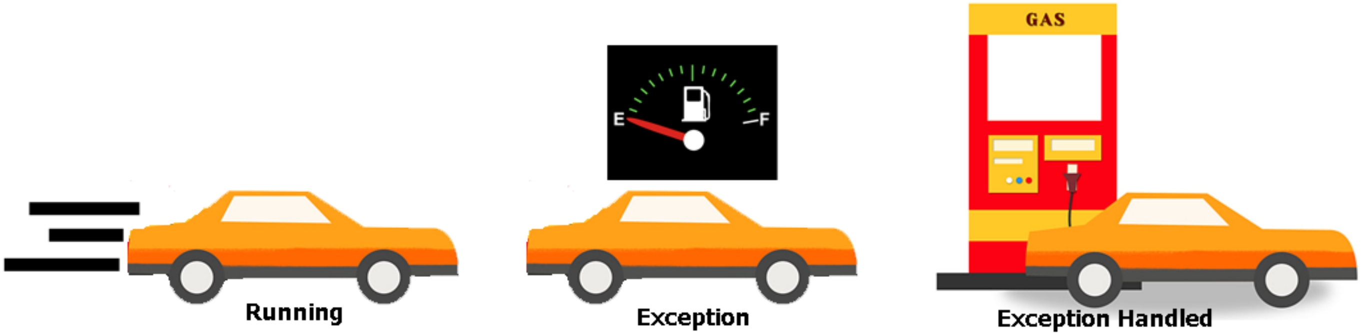
Must be caught
Must be declared
Recoverable errors

Exceptions

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

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.



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

- `ClassNotFound`
- `FileNotFound`
- `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***.

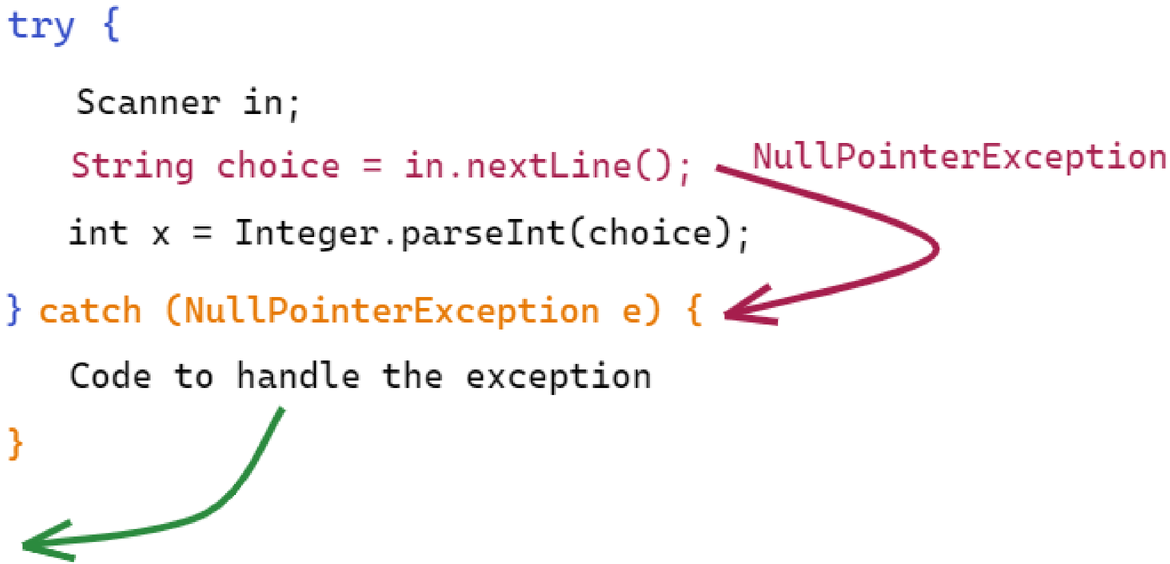
The *thrown* exception includes an *Exception Object* that contains details about what happened and a ***Stack Trace*** that details where it occurred in the code.

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.

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
}
```

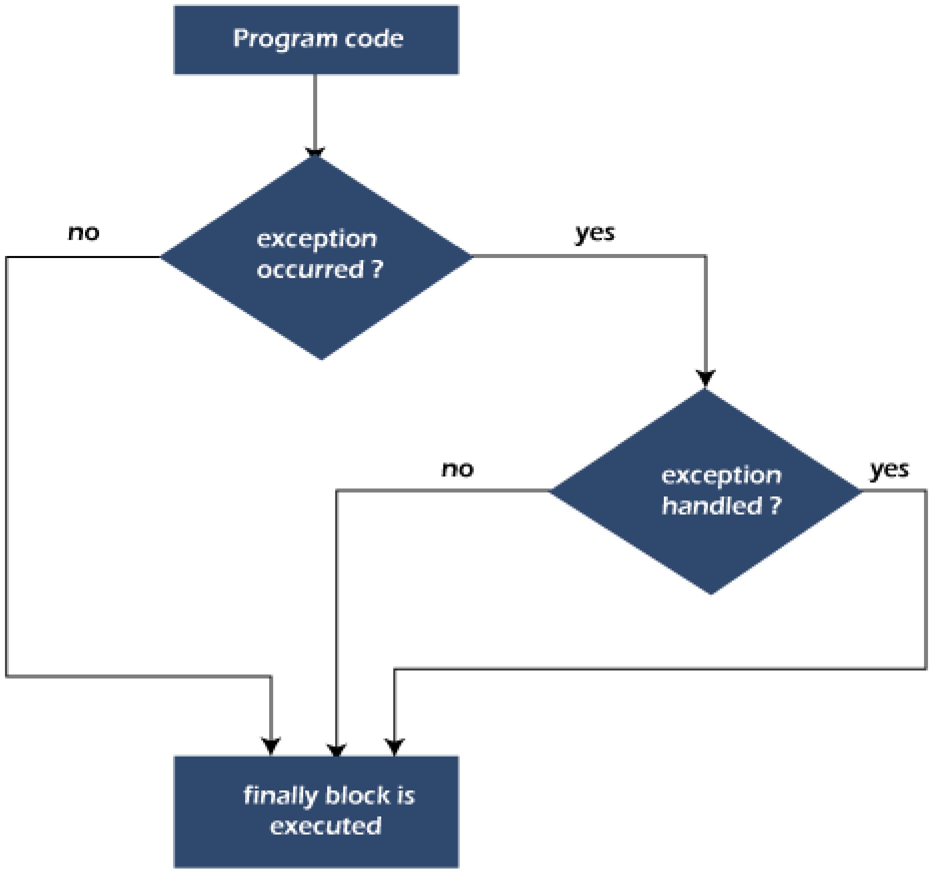
When an exception occurs in the try, all following lines of code are skipped and the catch is immediately executed.

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.



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.

Throw vs Throws

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

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

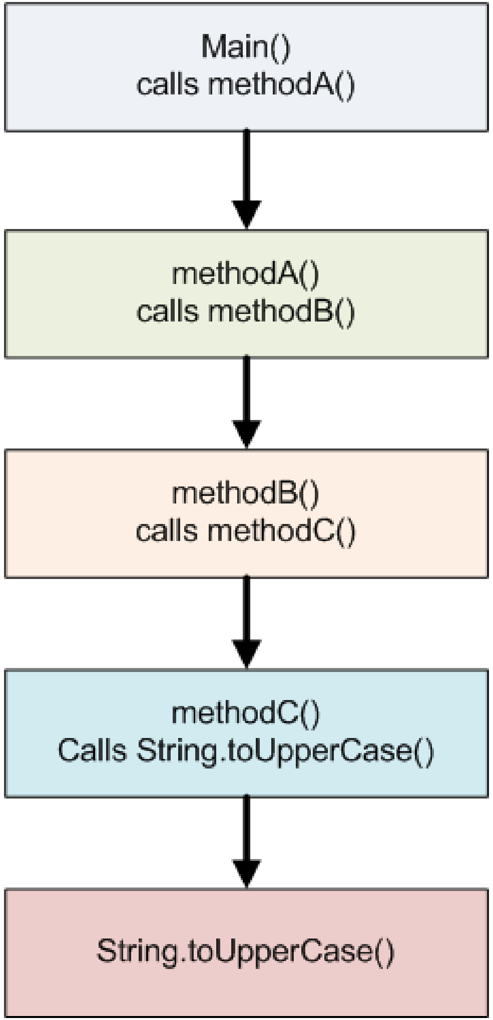
throws: A keyword used *in a method signature* to declare the method may throw an exception.

```java
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

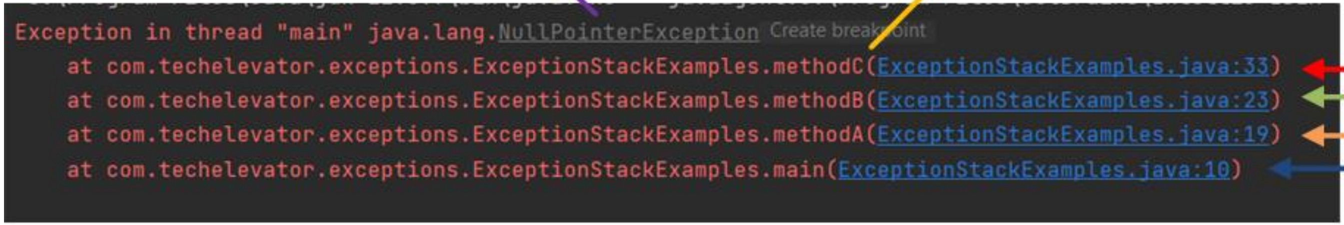
Methods call other methods. As each method is called it is added to the *Call Stack*, which is a map of what code is currently executing and the path the code took to get there.



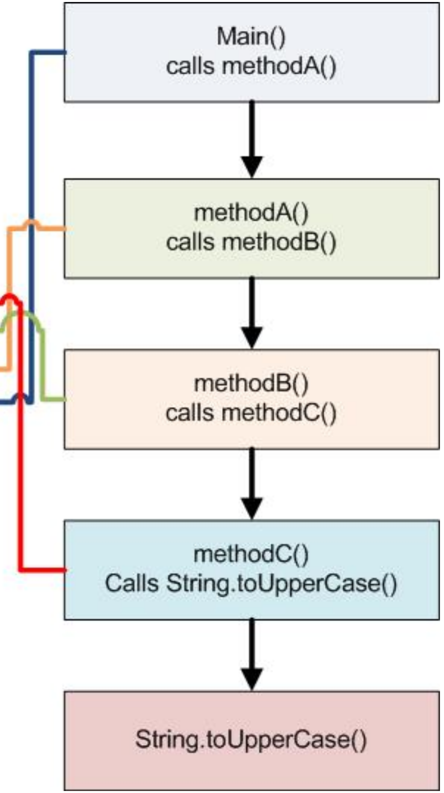
Stack Trace

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)
```

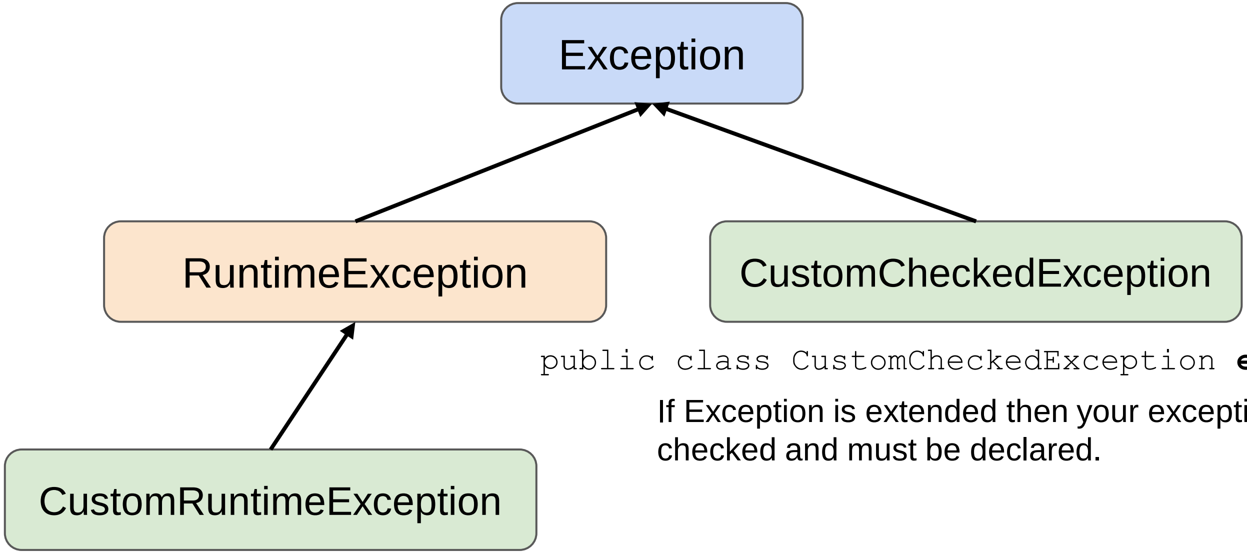


A stack trace is information included in an exception that shows the exception type thrown and call stack that existed when the exception occured. The call stack is read bottom to top.

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.



Exception

RuntimeException

CustomCheckedException

CustomRuntimeException

```
public class CustomCheckedException extends Exception
```

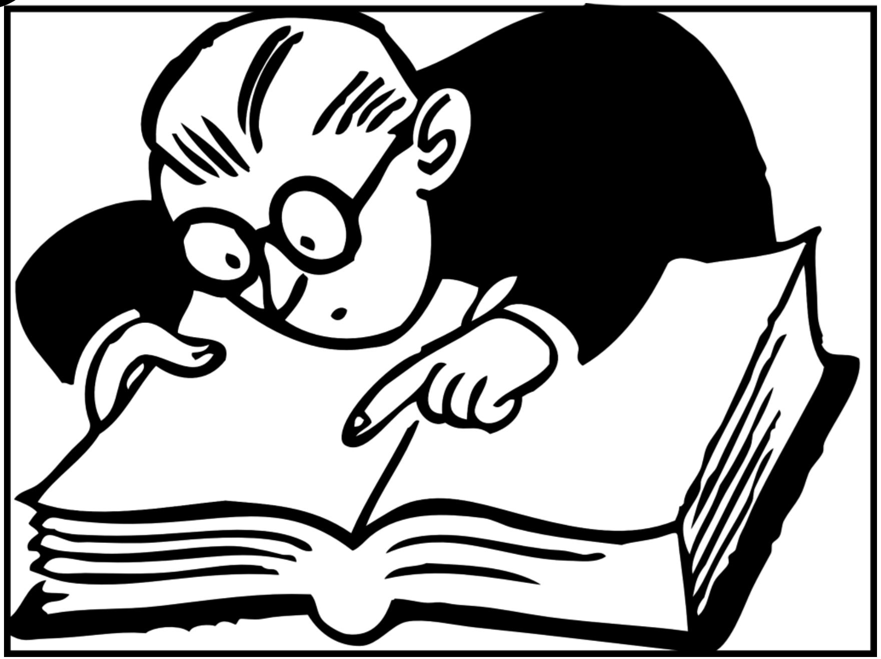
If Exception is extended then your exception will be checked and must be declared.

```
public class CustomRuntimeException extends RuntimeException
```

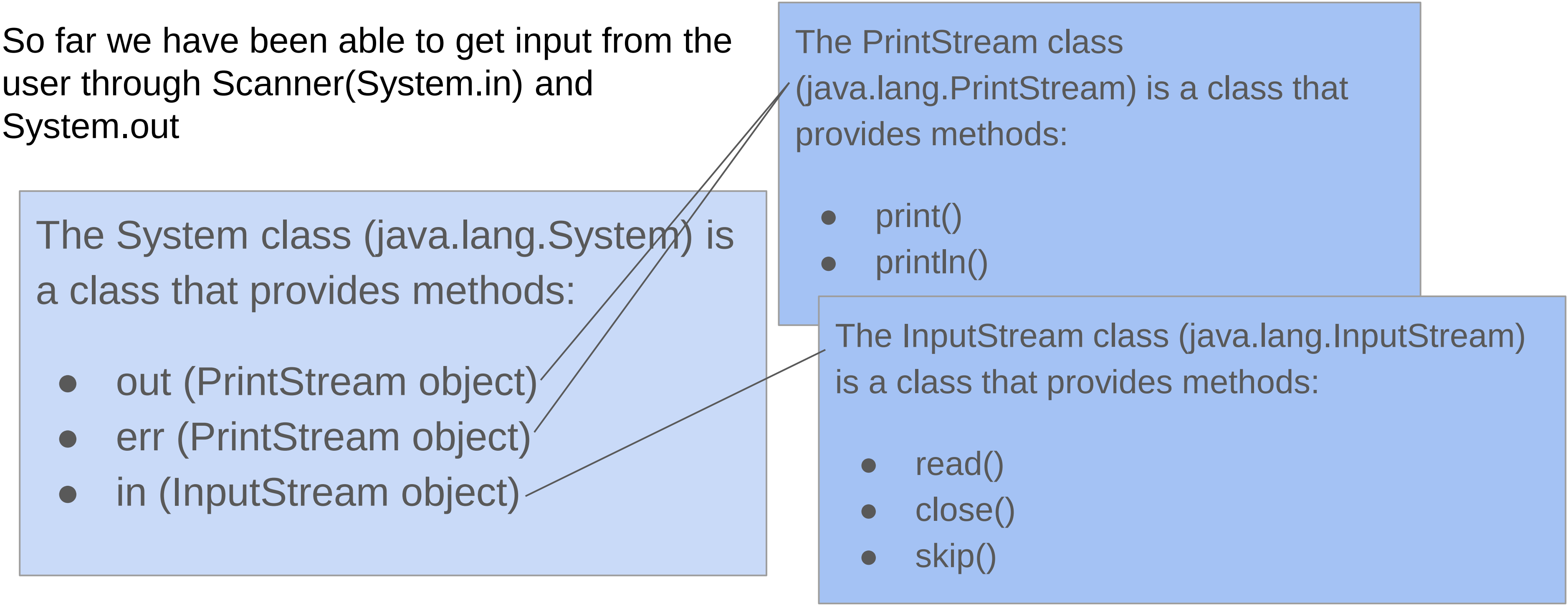
If RuntimeException is extended then your exception will be unchecked and does not need to be declared.

File IO

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:

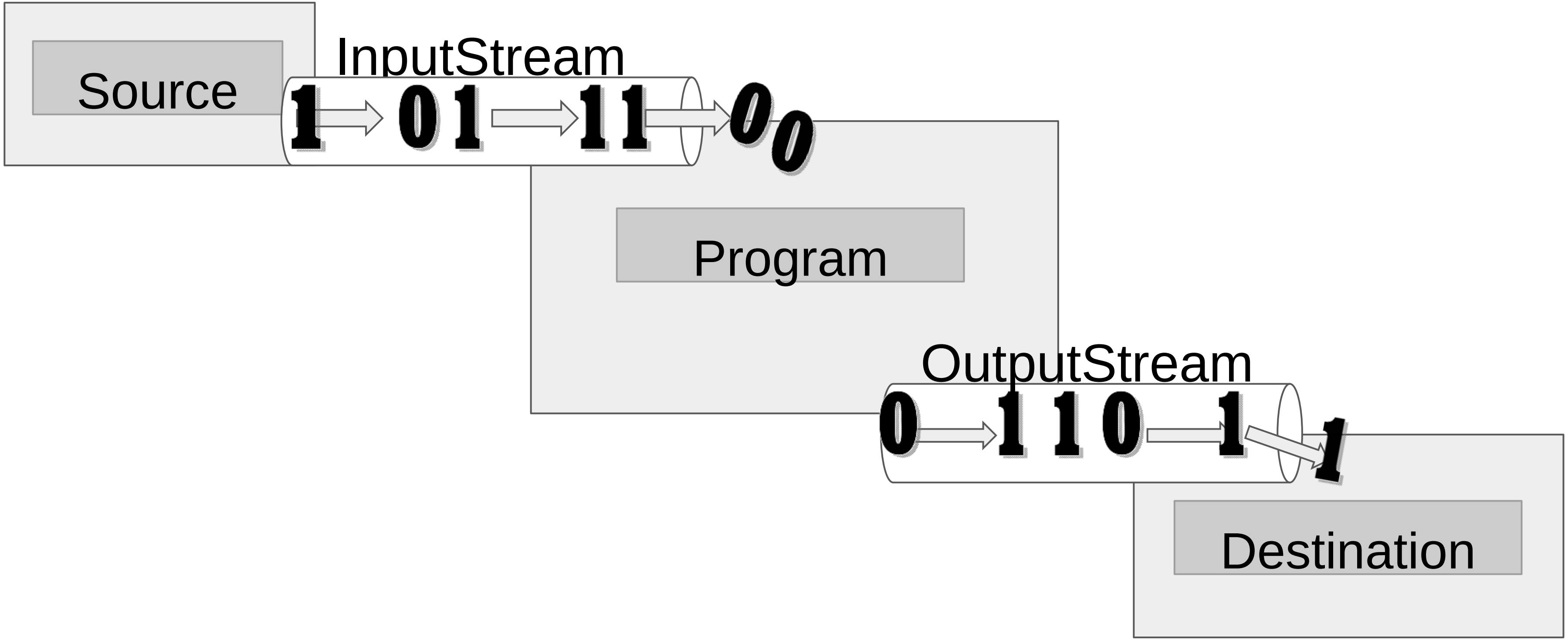
- print()
- println()

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)

A Stream refers to a sequence of bytes that can read and write to some sort of backing data store.



Java.io Library

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);
```

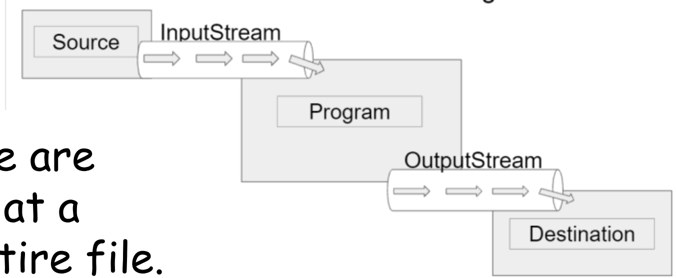
Note - File objects can only tell you information about the file. To open it, pass it to a Scanner object.

```
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.

Don't forget! We are reading one line at a time, not the entire file.

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.

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

Original:

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

    fileScanner.close();
```

Using try-with-resource:

```
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”);
    }
```