

Java System: Exceptions

**CPSC 233: Introduction to Computer Science for Computer Science
Majors II
Winter 2022**

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Monday, 8 November 2021

Copyright © 2021



Revisiting Errors

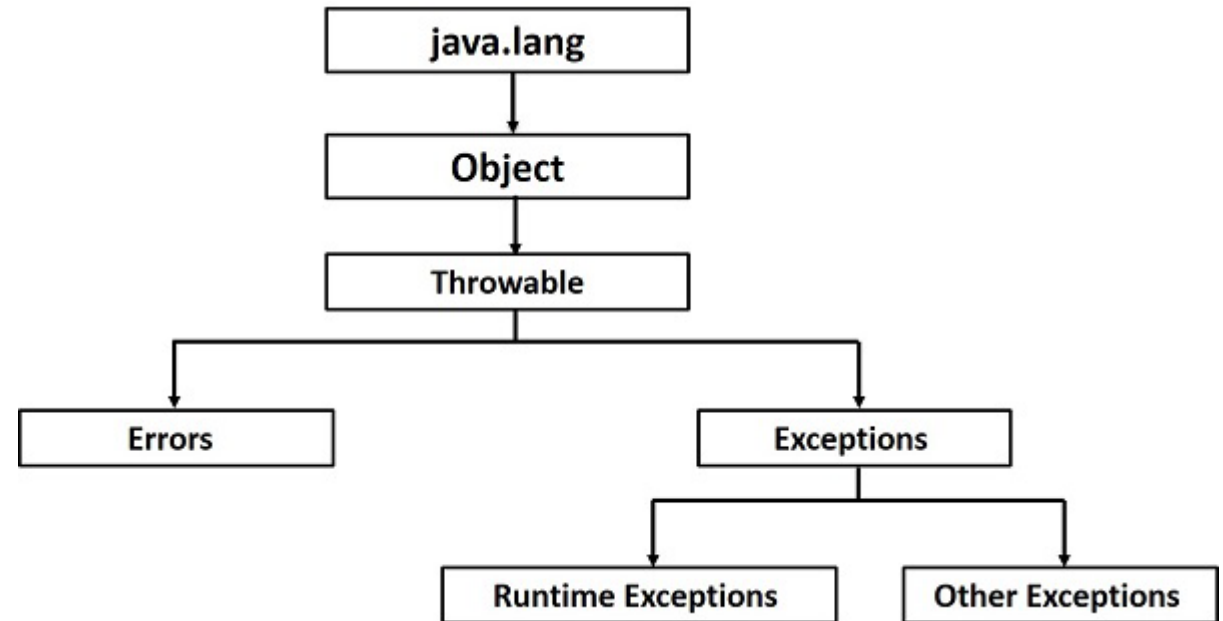
- Previously, you learned about the three main types of errors:
 1. **Syntax Errors:** refers to errors in the structure of a program and the rules about that structure.
 2. **Runtime Errors:** refers to errors that occur during program execution
 3. **Semantic/Logic Errors:** refers to errors in the logic of a program
- Runtime Errors are also referred to as ***Exceptions***

Exceptions

- An **exception** is an event that occurs during the execution of a program, which disrupts its execution.
- Exceptions can rise due to many reasons, including improper use of functions or operators, user input, logic errors, hardware and OS limitations, etc.
- Examples:
 - trying to access a list with an invalid index
 - trying to open a non-existent file
 - trying to parse a string using an invalid character
 - trying to converting a string to an integer
 - ...

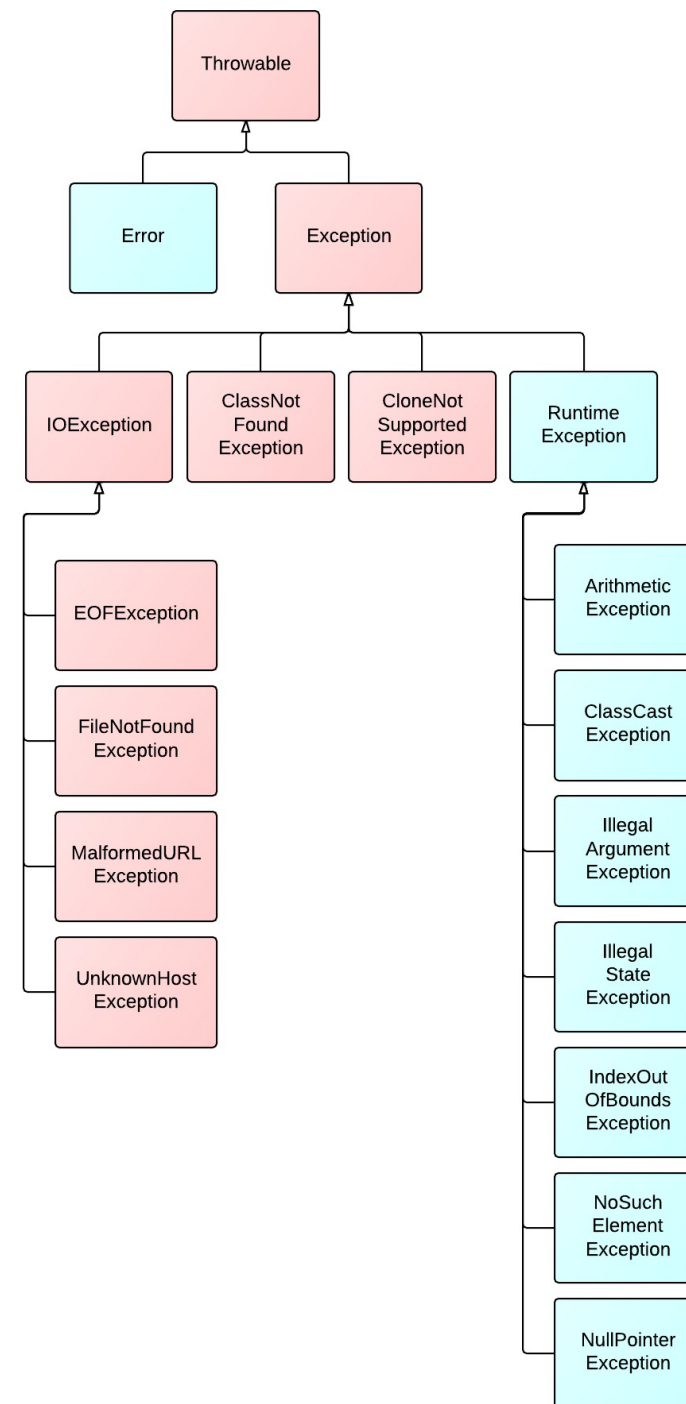
Exception Hierarchy

- All Exception are of Type Throwable
- Errors are for VM issues like OutOfMemory, NoSuchMethod
- https://www.tutorialspoint.com/java/lang/java_lang_errors.htm
- Exceptions are from things that your code does while running



Exception Hierarchy

- IOExceptions are encountered when dealing with files and other input/output libraries
- RuntimeExceptions are often from your internal methods
 - Invalid indices, math errors, data structure access errors, null pointer errors



Exceptions - Exception Handling

- Exceptions can be handled in several ways:
 - Using conditionals: the code handles scenarios where errors may occur.
 - Using **try/except** blocks: placing code that may fail within a try/except block.

Try/Except

Exceptions – Try/Except Block – (Python)

try: `<code segment that may cause error>` optional: can target certain types of exceptions

except (`<type>, <type>, ...`) as `<obj name>`: optional: a named exception object for accessing info on exception

`<action to take when an exception occurs>`

else: optional: executed only if no exceptions

`<action to take when no exception occurs>`

finally: optional: executed regardless of the code outcome

`<action to take in any case>`

Exceptions – Try/Except Block – (Java)

required: one
or more types
of exception to
target

```
try{  
    //<code segment causing error>  
} catch(<type> <obj_name>){  
    //<action to take>  
}
```

required: a named exception object
for accessing info on exception

Exceptions – Try/Except Block – (Java)

required: one or more types of exception to target

optional: exceptions to handle if earlier block doesn't

```
try{
    //<code segment causing error>
} catch(<type> | <type> | ... <obj_name>){
    //<action to take>
} catch(<type> | <type> | ... <obj_name>){
    //<action to take>
} finally{
    //<action to always take>
}
```

optional: executed regardless of the code outcome

Exceptions – Try/Except Block – (Java)

```
try {
    FileReader file_reader = new FileReader(file);
    BufferedReader buffered_reader = new BufferedReader(file_reader);
    String line = buffered_reader.readLine();
    while (line != "") {
        System.out.println(line);
        line = buffered_reader.readLine();
    }
} catch (FileNotFoundException e) {
    System.err.println("Could not find file: " + file.getAbsolutePath());
    System.exit(1);
} catch (IOException e) {
    System.err.println("Error reading from file: " + file.getAbsolutePath());
    System.exit(1);
}
```

Exceptions and Closing Files

Closing File – With Finally

```
FileReader file_reader = null;
try {
    file_reader = new FileReader(file);
    /// ...
} catch (FileNotFoundException e) {
    System.err.println("Could not find file: " + file.getAbsolutePath());
    System.exit(1);
} finally{
    try{
        file_reader.close();
    } catch (IOException e) {
    }
}
```

Closing File Using “with resources” Style

```
String filename = args[0];
File file = new File(filename);
try (FileReader file_reader = new FileReader(file); ){
    /// ...
} catch (FileNotFoundException e) {
    System.err.println("Could not find file: " + file.getAbsolutePath());
    System.exit(1);
} catch (IOException e) {
    System.err.println("Could not close file: " + file.getAbsolutePath());
    System.exit(1);
}
```

try (//with resources) {} - ensures that resource is closed at end of try

Closing File Using “with resources” Style

```
String filename = args[0];
File file = new File(filename);
try (FileReader file_reader = new FileReader(file);
    BufferedReader buffered_reader = new BufferedReader(file_reader);) {
    /// ...
} catch (FileNotFoundException e) {
    System.err.println("Could not find file: " + file.getAbsolutePath());
    System.exit(1);
} catch (IOException e) {
    System.err.println("Could not close file: " + file.getAbsolutePath());
    System.exit(1);
}
```

try (//with resources) {} - ensures that resources are closed at end of try

Debugging

Produce the stack trace from program crashing

- If you want the stack trace you usually saw (for debugging purposes)
- Use `e.printStackTrace()`

```
String filename = args[0];
File file = new File(filename);
try (FileReader file_reader = new FileReader(file);
    BufferedReader buffered_reader = new BufferedReader(file_reader);) {
    /// ...
} catch (IOException e) {
    e.printStackTrace();
    System.err.println("Could not close file: " + file.getAbsolutePath());
    System.exit(1);
}
```

- `e.getMessage()` also useful for accessing previous exception message

Throwing Exceptions

Other Exception Handling

1. Functions Throwing Exceptions

- When you discover error, create an exception object and pass it to the system. Two steps:

1. header says it might throw an exception

```
public void amethod() throws IllegalArgumentException
```

2. if an error occurs, create the exception object and throw it

```
throw new IllegalArgumentException ("Invalid function parameter");
```

Example of throw processing

```
public static double div(double x, double y) throws IllegalArgumentException{  
    if (y == 0){  
        throw new IllegalArgumentException ("Can't divide by 0");  
    }  
    return x / y;  
}
```

Your Own Exceptions?

This will be clearer when we reach Classes and Objects

```
//*****  
// OutOfRangeException.java    Java Foundations  
//  
// Represents an exceptional condition in which a value is out of  
// some particular range.  
//*****  
  
public class OutOfRangeException extends Exception  
{  
    // Sets up the exception object with a particular message.  
    public OutOfRangeException (String message)  
    {  
        super (message);  
    }  
}
```

```
public static void main (String[] args) throws OutOfRangeException {
    final int MIN = 25, MAX = 40;
    Scanner scan = new Scanner (System.in);
    OutOfRangeException problem = new OutOfRangeException ("Input value is out of range.");
    System.out.print ("Enter an integer value between " + MIN + " and " + MAX + " : ");
    int value = scan.nextInt();
    // Determine if the exception should be thrown
    if (value < MIN || value > MAX)
        throw problem;
    System.out.println ("End of main method."); // may never reach
}
}
```

Onward to ... Classes and Objects.

Jonathan Hudson
jwhudson@ucalgary.ca
<https://pages.cpsc.ucalgary.ca/~jwhudson/>

