26-Exception

Author: Vincent Lau

Learning Objectives

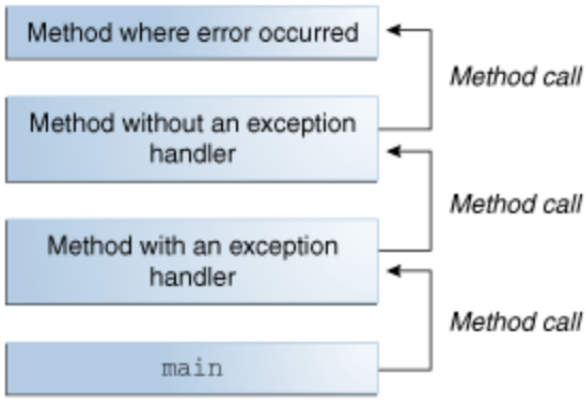
> Understand the exception hierarchy and how exceptions work in Java
>
> Describe the difference between Checked Exceptions, Unchecked Exceptions, and Errors
>
> Use various techniques to catch and handle exceptions
>
> Use *try-with-resources* to replace the traditional cleanup approach with *finally*
>
> Create custom exceptions by extending the Exception class

Overview

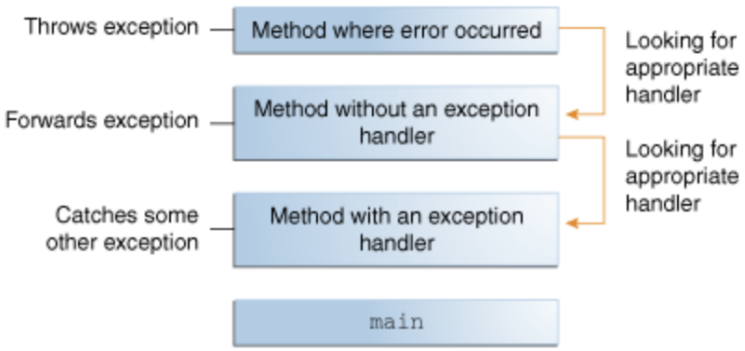
What is an Exception?

- The term *exception* is shorthand for the phrase "exceptional event."

- **Definition** - An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- For each method call, a stack frame will be created and pushed onto the top of the stack, which is called the **call stack**.



- When an error occurs within a method, the method creates an *exception object*, and *throws an exception* to the runtime system. The *exception object* contains information about the error, including its type and the state of the program when the error occurred.
- The runtime system searches the call stack for a method that contains a block code that can handle the exception, which is called an *exception handler*.
- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler to *catch the exception*, the whole program terminates.



Why use Exceptions?

- In production systems, bad things happen - such as corrupted files, broken down network, and an out-of-memory JVM. While we very much like the code to run in the "happy paths", we must also write code to handle erroneous conditions in the "unhappy paths".

```
1 public class StackTraceDemo {
2     public static void main(String[] args) { // throws IOException
3         Path filePath = Paths.get("someFile.txt");
4         List<String> lines = Files.readAllLines(filePath);
5         System.out.println(lines);
```

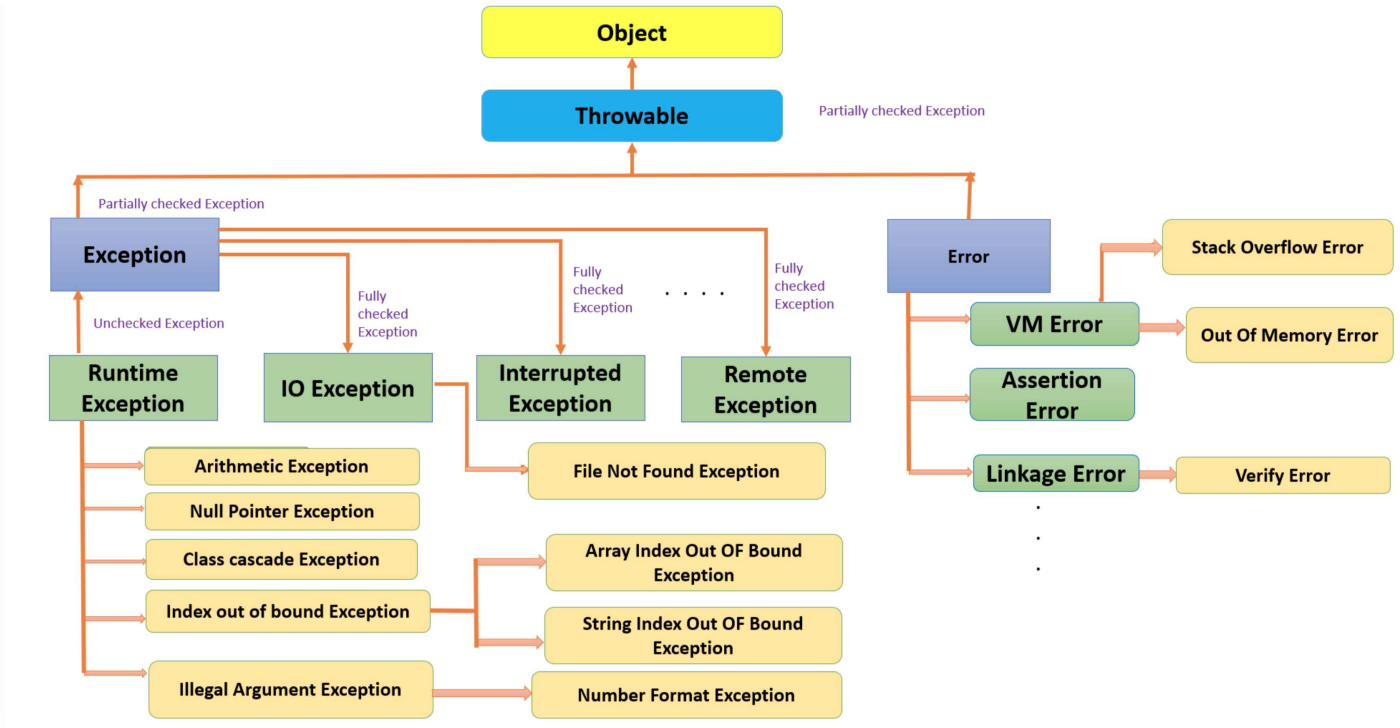
```
    }
}
```

- The code above decides not to handle the IOException. What if there is no such file?

```
Exception in thread "main" java.nio.file.NoSuchFileException: someFile.txt
        at java.base/sun.nio.fs.WindowsException.translateToIOException(WindowsException.java:85)
        at java.base/sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:103)
        at java.base/sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:108)
        at java.base/sun.nio.fs.WindowsFileSystemProvider.newByteChannel(WindowsFileSystemProvider.java:231)
        at java.base/java.nio.file.Files.newByteChannel(Files.java:370)
        at java.base/java.nio.file.Files.newByteChannel(Files.java:421)
        at java.base/java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.java:420)
        at java.base/java.nio.file.Files.newInputStream(Files.java:155)
        at java.base/java.nio.file.Files.newBufferedReader(Files.java:2838)
        at java.base/java.nio.file.Files.readAllLines(Files.java:3327)
        at java.base/java.nio.file.Files.readAllLines(Files.java:3367)
        at Exceptions.StackTraceDemo.main(StackTraceDemo.java:13)
```

- Without handling the exception, the program may stop running altogether.
- The bright side is - we have the stack trace above to help us debug what might have gone wrong.

Exception Hierarchy



- In Java, all exceptions extend from *Throwable.*
- There are three main types of exceptional conditions:
  - Unchecked Exception (also known as Runtime Exceptions)
  - Checked Exceptions
  - Errors

Checked Exceptions

- **Checked exceptions** are exceptions that the **Java compiler** requires us to handle. We have to either declaratively throw the exception up the call stack, or we have to handle it in the current method call.
- We use checked exceptions when we reasonably expect the caller of the method to be **able to recover**.
- All exceptions are checked exceptions, except for those indicated by *Error, RuntimeException,* and their subclasses.
- Examples include *IOException* and *SQLException.*

Unchecked Exceptions

- Unchecked exceptions are those that are **internal** to the application, and that the application cannot anticipate or recover from.
- Unchecked exceptions are exceptions that the Java compiler does *not* require us to handle. (Compiler somehow expects us to **avoid** the exception scenario, for example, divide 0/ null pointer)

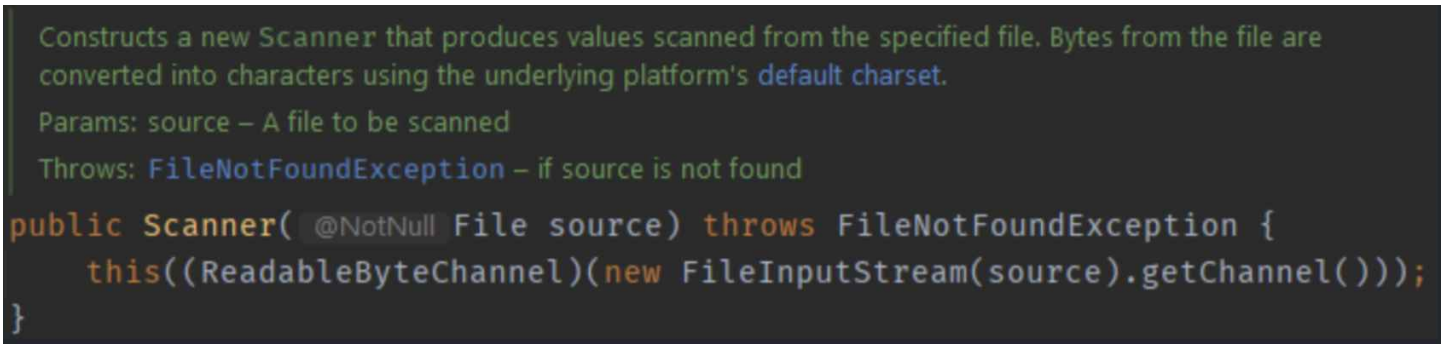
- If we create an exception that extends *RuntimeException*, it is considered an unchecked exception.
- Examples include *ArrayIndexOutOfBoundsException, NullPointerException* and *IllegalArgumentException.*

Errors

- These are exceptional conditions that are **external** to the application, and that the application usually cannot anticipate or recover from, such as library incompatibility, infinite recursion, or memory leaks.
- Examples include *NoClassDefFoundError, StackOverflowError* and *OutOfMemoryError.*

Handling Exceptions

- In the Java API, there are plenty of places where checked exceptions and unchecked exceptions can be thrown.
- It is a ***must*** to handle checked exceptions (i.e. FileNotFoundException), and it is *optional* to handle unchecked exceptions.



- There are multiple ways we can handle exceptions.

throws

- The simplest way to "handle" an exception is to rethrow the exception.

```
1 public int getSomeInteger(String filePath) throws FileNotFoundException {
2     try {
3         Scanner scanner = new Scanner(new File(filePath));
4         // n lines of code ...
5     } catch (FileNotFoundException e) {
6         // ....
7     }
8
```

```
    return Integer.parseInt(scanner.nextLine());
}

void main {
    getSomeInteger("E:\\test.txt"); // error
}
```

- By **adding the exception to the method signature,** we declaratively rethrow the exception up the call stack.
- We did not actually handle the exception, but this is the simplest way to satisfy the compiler.
- *parseInt* can throw a *NumberFormatException,* but because it is unchecked, we are not required to handle it.

try-catch

- If we want to try and handle the exception ourselves, we can use a *try-catch* block.
- We can either:
  a. **Rethrow a higher-level exception to a higher-level handler** (useful when a service can throw multiple types of errors and we want to wrap them all in one higher-level exception)

```
public int getSomeInteger(String filePath) {
    try {
        Scanner scanner = new Scanner(new File(filePath));
        return Integer.parseInt(scanner.nextLine());
    } catch (FileNotFoundException e) {
        throw new FileServiceException("File not found", e);
    }
}
```

1. Or **perform recovery steps by returning something**

```
public int getSomeInteger(String filePath) {
    try {
        Scanner scanner = new Scanner(new File(filePath));
        return Integer.parseInt(scanner.nextLine());
    } catch (FileNotFoundException e) {
        System.out.println("File not found! Returning 0...");
        return 0;
    }
}
```

finally

- Sometimes we want to execute some code (such as cleanup) regardless of whether an exception occurs, and this is where the ***finally*** keyword comes in.
- When we perform operations with external resources (such as the filing system or the operating system), it is always a good practice to remember to clean up the resources after using them.

```
1  public int getSomeInteger(String filePath) throws FileNotFoundException {
2      Scanner scanner = null;
3      try {
4          scanner = new Scanner(new File(filePath));
5          return Integer.parseInt(scanner.nextLine());
6      } finally {
7          if (scanner != null) {
8              scanner.close();
9          }
10     }
11 }
```

- Indeed, we can handle the exception *and* make sure that our resources get cleaned up.
- The *try* block can be followed by either the *catch* or *finally* block, or both.

```
1  public int getSomeInteger(String filePath) {
2      Scanner scanner = null;
3      try {
4          scanner = new Scanner(new File(filePath));
5          return Integer.parseInt(scanner.nextLine());
6      } catch (FileNotFoundException e) {
7          System.out.println("File not found! Returning 0...");
8          return 0;
9      } finally {
10         if (scanner != null) {
11             scanner.close();
12         }
13     }
14 }
```

try-with-resources

- Ever since Java 7, we can simplify *finally-cleanup* logic when working with classes that implements *the AutoCloseable* interface. Scanner is one of these classes.

```java
public int getSomeInteger(String filePath) {
    try (Scanner scanner = new Scanner(new File(filePath))) {
        return Integer.parseInt(scanner.nextLine());
    } catch (FileNotFoundException e) {
        System.out.println("File not found! Returning 0...");
        return 0;
    }
}
```

- When we place references that are ***AutoClosable*** in the try declaration, we do not need to close the resource ourselves.
- Indeed, we can still use a ***finally* block**, if we want to perform some **additional cleanup**.

Multiple *catch* Blocks

- There are times the code can throw multiple exceptions, and we want to handle them with more than one *catch* block.
- Multiple *catch* blocks allow us to handle each exception differently if needed.

```java
public int getSomeInteger(String filePath) {
    try (Scanner scanner = new Scanner(new File(filePath))) {
        // Some other logic that may throw IOException
        return Integer.parseInt(scanner.nextLine());
    } catch (FileNotFoundException e) {
        System.out.println("File not found! Returning 0...");
        return 0;
    } catch (IOException e) {
        System.out.println("Error interacting with file! Returning 0...");
        return 0;
    } catch (NumberFormatException e) {
        System.out.println("Unable to parse integer! Returning 0...");
        return 0;
    }
}
```

- Notice *FileNotFoundException* extends *IOException*, **always try to catch the more specific exceptions first** (i.e. *FileNotFoundException* in this case).

- If we do not intend to handle *FileNotFoundException* and *IOException* differently, we could have just caught the *IOException,* because any of its subclasses will also have been caught.

Union *catch* Blocks

- Java 7 introduced the ability to **catch mutiple exceptions in the same block**, which is particularly useful if we intend to handle all exceptions the same way.
- Notice the `|` operator that separates the exceptions.

```
1 public int getSomeInteger(String filePath) {
2     try (Scanner scanner = new Scanner(new File(filePath))) {
3         return Integer.parseInt(scanner.nextLine());
4     } catch (IOException | NumberFormatException e) {
5         System.out.println("Error with file! Returning 0...");
6         return 0;
7     }
8 }
```

Throwing Exceptions

Creating Your Own Exception

- Sometimes, you want to write your own exception to represent a specific type of errors related to a component in your application.
- It is good practice to append the string *Exception* to your own Exception class.
- You can pass the error message and/or the original error that is being thrown to the higher-level exception you created.

```
1 public class PizzaOrderingException extends Exception {
2     public PizzaOrderingException(String msg) {
3         super(msg);
4     }
5
6     public PizzaOrderingException(String msg, Throwable e) {
7         super(msg, e);
8     }
9 }
```

Throwing a Checked Exception

```
public class PizzaOrderingService {
    // method
    public void order(String orderId) throws CustomException {
        while (!timeOutThresholdExceeded()) {
            // process order
            // somewhere change the value of timeOutThresholdExceeded
            return;
        }
        // or use try catch
        try {
            throw new PizzaOrderingException("The ordering took too long!");
        } catch (PizzaOrderingException e) {
            // nothing
            throw new CustomException();
        }
    }

    void main() {
        try {
            order();
        } catch (IllegalArgumentException e) {
            order("abc");
        }
    }
}
```

Throwing an Unchecked Exception

```
public void order(String orderId) {
    if (orderId == null || orderId.isEmpty()) { // ""
        throw new IllegalArgumentException("Invalid orderId !!");
    }

    // remaining code
}
```

Chained Exceptions

- An application often responds to an exception by throwing another exception. In effect, the first exception *causes* the second exception. It can be very helpful to know when one exception causes another. This is where *Chained Exceptions* come in.
- In the example, when an *IOExeption* is caught, a new *PizzaOrderingException* is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

```
1  public void order(String orderId) throws PizzaOrderingException {
2            // some validation code
3
4      try {
5          interactWithExternalResource();
6      } catch (IOException e) {   // file not found
7          throw new PizzaOrderingException("Error occurred with external
   resource", e);
8      }
9
10                 // some other logic
11 }
```

Questions

- What are the difference between Checked Exceptions, Unchecked Exceptions, and Errors?
- Describe the different ways we can catch one or multiple exceptions.
- How does *try-with-resources* work?
- How to write your own Exception class?

Reading Exercise

ArithmeticException

- It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
1  // Java program to demonstrate ArithmeticException
2  class ArithmeticExceptionDemo
3  {
4      public static void main(String args[])
```

```
    {
        try {
            int a = 30, b = 0;
            int c = a / b;  // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 0");
        }
    }
} // Output: Can't divide a number by 0
```

ArrayIndexOutOfBoundsException

- It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

```
 // Java program to demonstrate ArrayIndexOutOfBoundException
class ArrayIndexOutOfBoundDemo
{
    public static void main(String args[])
    {
        try {
            int a[] = new int[5];
            a[6] = 9; // accessing 7th element in an array of
                      // size 5
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println ("Array Index is Out Of Bounds");
        }

    }
} // Output: Array Index is Out Of Bounds
```

ClassNotFoundException

- This Exception is raised when we try to access a class whose definition is not found

```
// Java program to demonstrate ClassNotFoundException
public class ClassNotFoundException_Demo
{
    public static void main(String[] args) {
        try{
```

```
            Class.forName("Class1");   // Class1 is not defined
        }
        catch(ClassNotFoundException e){
            System.out.println(e);
            System.out.println("Class Not Found...");
        }
    }
}
// Output:
// java.lang.ClassNotFoundException: Class1
// Class Not Found...
```

FileNotFoundException

- This Exception is raised when a file is not accessible or does not open.

```
//Java program to demonstrate FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
 class File_notFound_Demo {

    public static void main(String args[])  {
        try {
            // this line tries to find the file.txt in E drive
            // Following file does not exist
            File file = new File("E://file.txt");

            FileReader fr = new FileReader(file);

        } catch (FileNotFoundException e) {
           System.out.println("File does not exist");
        }
    }
} // Output: File does not exist
```

IOException

- It is thrown when an input-output operation failed or interrupted

```
 // Java program to demonstrate IOException
class IOExceptionDemo {

```

```
    public static void main(String[] args)
    {

        // Create a new scanner with the specified String
        // Object
        Scanner scan = new Scanner("Hello Geek!");

        // Print the line
        System.out.println("" + scan.nextLine());

        // Check if there is an IO exception
        System.out.println("Exception Output: "
                           + scan.ioException());

        scan.close();
    }
}
// Output:
// Hello Geek!
// Exception Output: null
```

NullPointerException

- This exception is raised when referring to the members of a null object. Null represents nothing

```
//Java program to demonstrate NullPointerException
class NullPointerDemo
{
    public static void main(String args[])
    {
        try {
            String a = null; //null value
            char b = a.charAt(0);
        } catch(NullPointerException e) {
            System.out.println("NullPointerException..");
        }
    }
} // Output: NullPointerException..
```

NumberFormatException

- This exception is raised when a method cannot convert a string into a numeric format.

```java
// Java program to demonstrate NumberFormatException
class  NumberFormat_Demo
{
    public static void main(String args[])
    {
        try {
            // "akki" is not a number
            int num = Integer.parseInt ("akki") ;

            System.out.println(num);
        } catch(NumberFormatException e) {
            System.out.println("Number format exception");
        }
    }
} // Output: Number format exception
```

StringIndexOutOfBoundsException

- It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

```java
// Java program to demonstrate StringIndexOutOfBoundsException
class StringIndexOutOfBoundDemo
{
    public static void main(String args[])
    {
        try {
            String a = "This is like chipping "; // length is 22
            char c = a.charAt(24); // accessing 25th element
            System.out.println(c);
        }
        catch(StringIndexOutOfBoundsException e) {
            System.out.println("StringIndexOutOfBoundsException");
        }
    }
} // Output: StringIndexOutOfBoundsException
```

IllegalArgumentException

- This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under unchecked exception.

```java
/*package whatever //do not write package name here */
import java.io.*;

class IaeDemo {
   public static void print(int a)
    {
         if(a >= 18) {
              System.out.println("Eligible for Voting");
          } else {
              throw new IllegalArgumentException("Not Eligible for Voting");

          }

    }
    public static void main(String[] args) {
         IaeDemo.print(14); // IAE
    }
}
// Output:
// Exception in thread "main" java.lang.IllegalArgumentException: Not Eligible for Voting
// at IaeDemo.print(File.java:13)
// at IaeDemo.main(File.java:19)
```

IllegalStateException

- This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under unchecked exception.

```java
import java.io.*;

class IseDemo {
      public static void  print(int a,int b) {
         System.out.println("Addition of Positive Integers :"+(a+b));
     }

    public static void main(String[] args) {
        int n1 = 7;
        int n2 = -3;

         if(n1 >= 0 && n2 >= 0) {
             IseDemo.print(n1, n2);
         } else {
             throw new IllegalStateException("Either one or two numbers are not Positive Integer");
```

```
        }
    }
}
// Output:
// Exception in thread "main" java.lang.IllegalStateException: Either one or two numbers are not Positive Integer
// at IseDemo.main(File.java:20)
```