Exceptions and Exception Handling

Fall 2022

By Pakita Shamoi

“Everything in Java in as Object and Exception is not an exception”

Pakita

So, some exceptions are subclasses of the others.

For example, **FileNotFoundException** is a child class of **IOException**. If you catch **IOException**, you also catch all its child exceptions, including **FileNotFoundException**

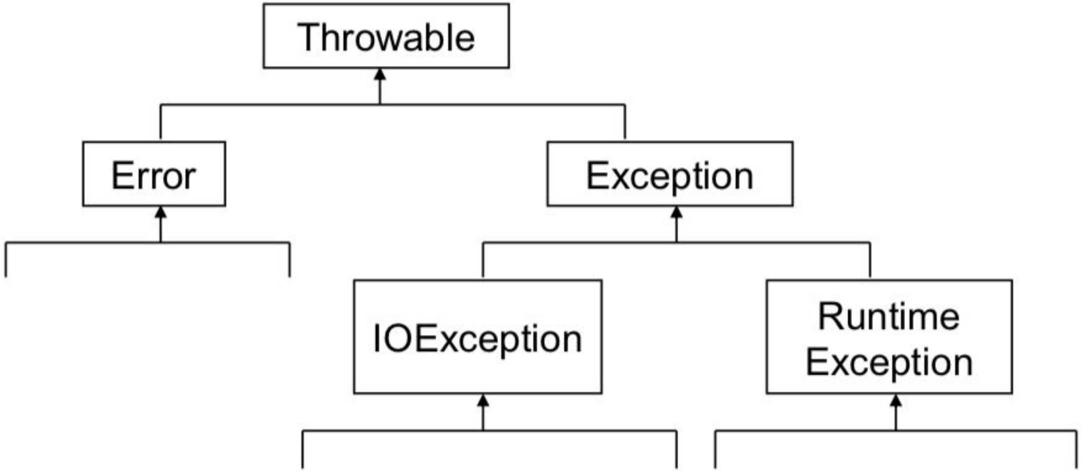
Definitions

Exception - an event that occurs during the execution of a program that disrupts the normal flow of instructions” is called an exception. This is generally an unexpected or unwanted event which can occur either at compile-time or run-time in application code.

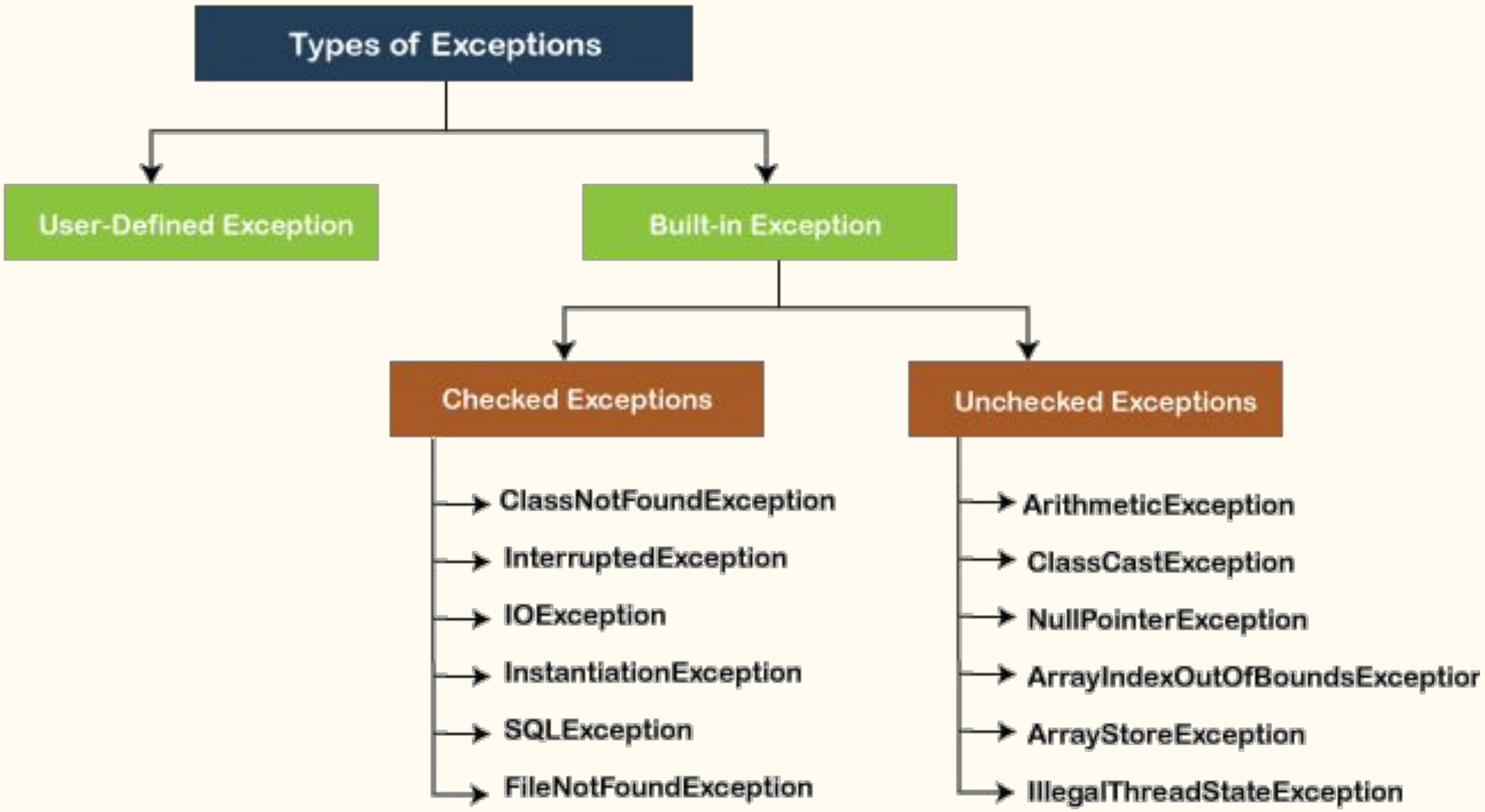
Exception Handling - actions that need to be done in case exception happens, in order to recover from it and continue execution. No handling = program execution is aborted.

Overview of the exception hierarchy

A simplified diagram of the exception hierarchy in Java



- All exceptions extend the class `Throwable`, which immediately splits into two branches: `Error` and `Exception`
  - □ **`Error:`** internal errors and resource exhaustion inside the Java runtime system. Little you can do.
  - □ **`Exception:`** splits further into two branches.



Types of Exceptions

User-Defined Exception

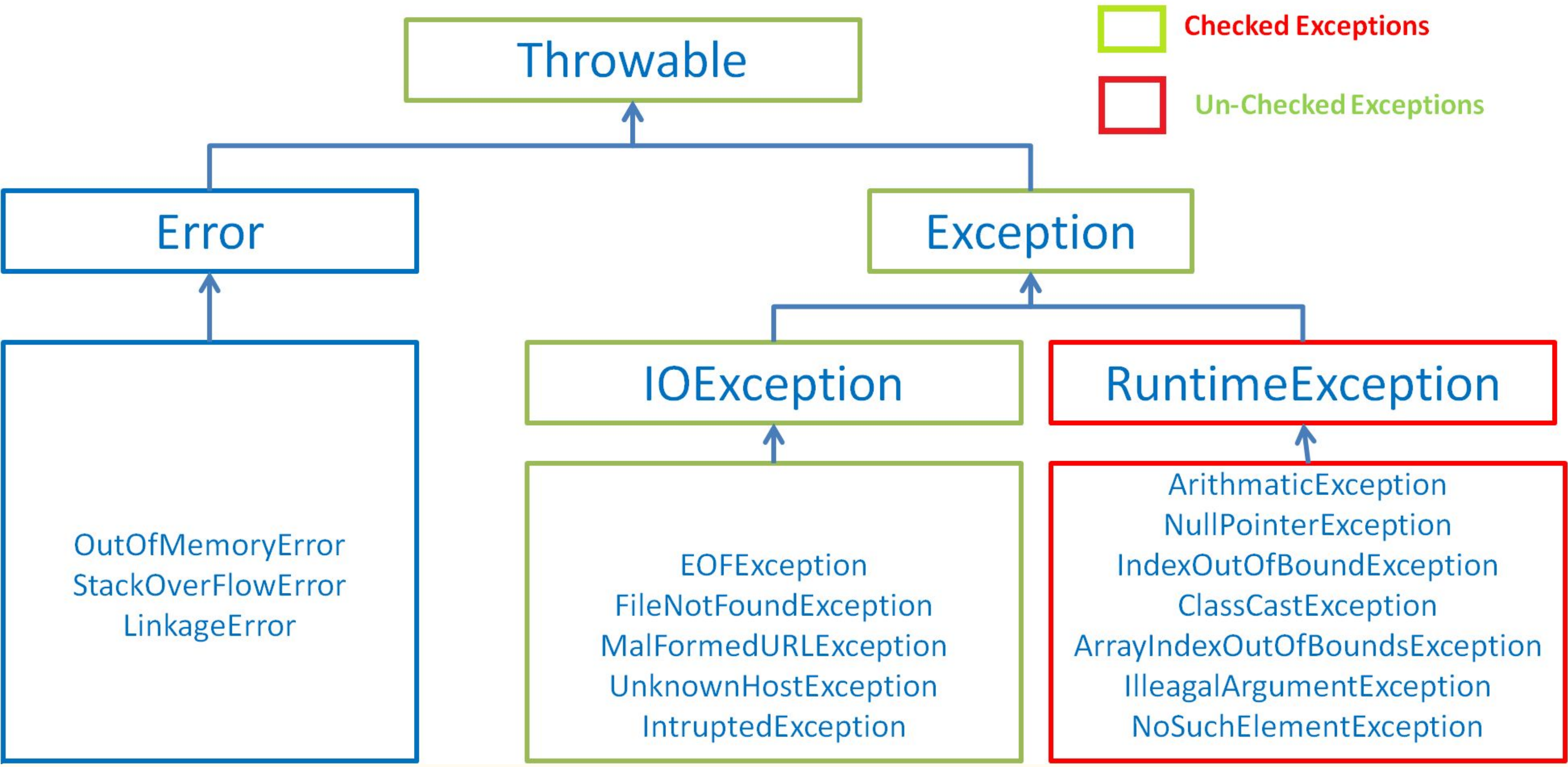
Built-in Exception

Checked Exceptions

- ClassNotFoundException
- InterruptedException
- IOException
- InstantiationException
- SQLException
- FileNotFoundException

Unchecked Exceptions

- ArithmeticException
- ClassCastException
- NullPointerException
- ArrayIndexOutOfBoundsExceptior
- ArrayStoreException
- IllegalThreadStateException



🟩 Checked Exceptions

🟥 Un-Checked Exceptions

Throwable

Error

Exception

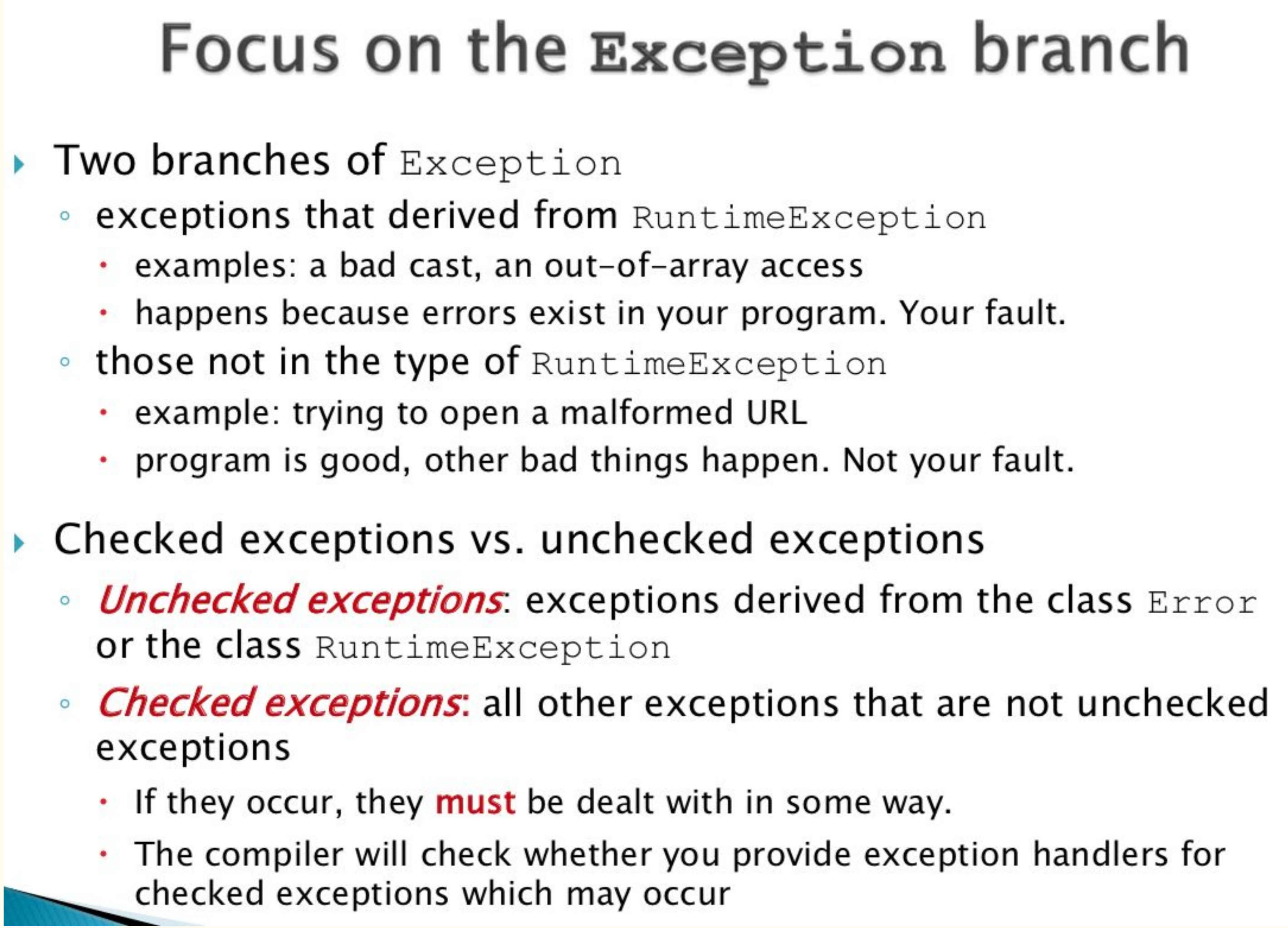
IOException

RuntimeException

OutOfMemoryError
StackOverFlowError
LinkageError

EOFException
FileNotFoundException
MalFormedURLException
UnknownHostException
IntruptedException

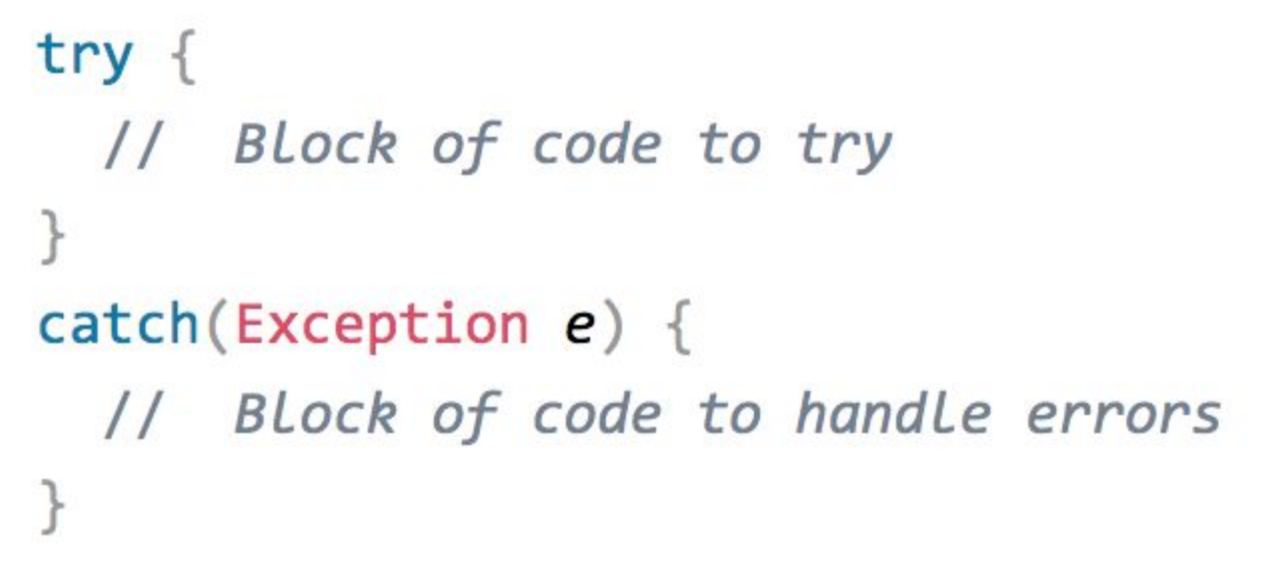
ArithmaticException
NullPointerException
IndexOutOfBoundException
ClassCastException
ArrayIndexOutOfBoundsException
IlleagalArgumentException
NoSuchElementException



Focus on the `Exception` branch

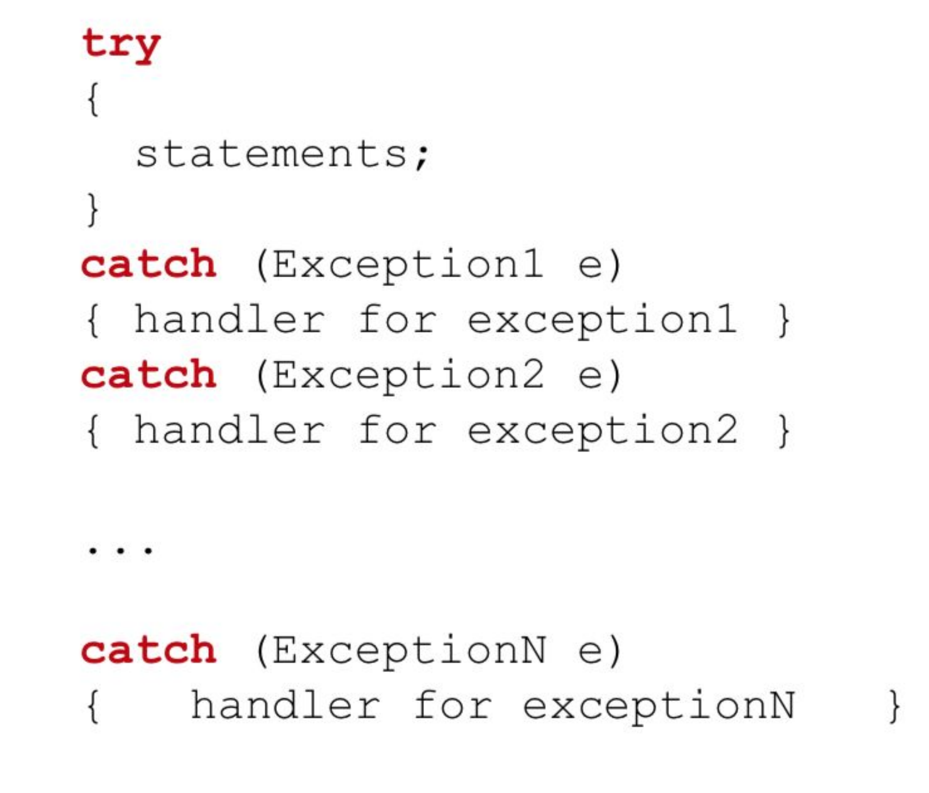
- Two branches of `Exception`
  - exceptions that derived from `RuntimeException`
    - examples: a bad cast, an out-of-array access
    - happens because errors exist in your program. Your fault.
  - those not in the type of `RuntimeException`
    - example: trying to open a malformed URL
    - program is good, other bad things happen. Not your fault.
- Checked exceptions vs. unchecked exceptions
  - ***Unchecked exceptions***: exceptions derived from the class `Error` or the class `RuntimeException`
  - ***Checked exceptions***: all other exceptions that are not unchecked exceptions
    - If they occur, they **must** be dealt with in some way.
    - The compiler will check whether you provide exception handlers for checked exceptions which may occur

Try Catch

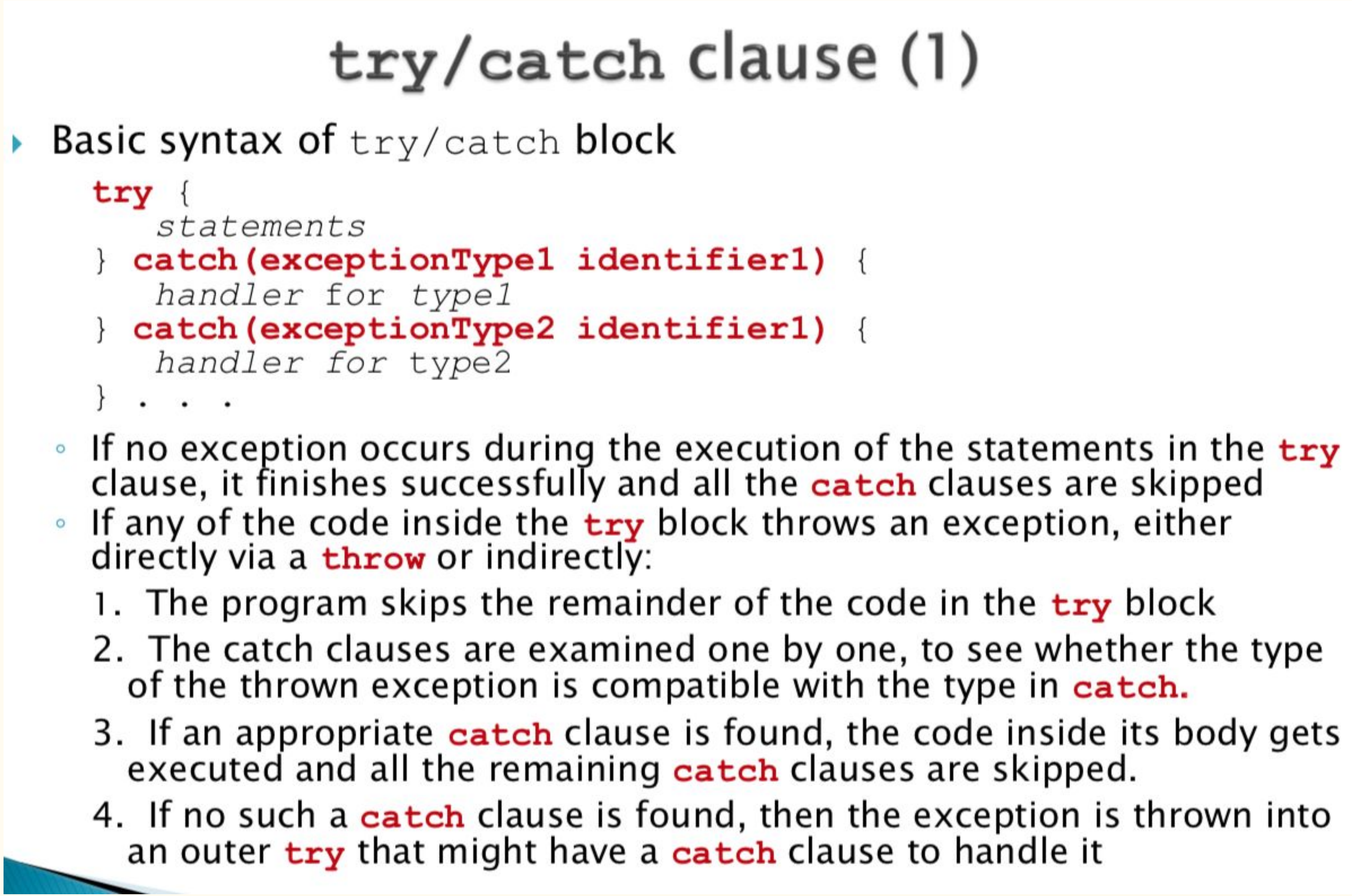


```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Try Catch



```
try
{
  statements;
}
catch (Exception1 e)
{ handler for exception1 }
catch (Exception2 e)
{ handler for exception2 }

...

catch (ExceptionN e)
{   handler for exceptionN   }
```



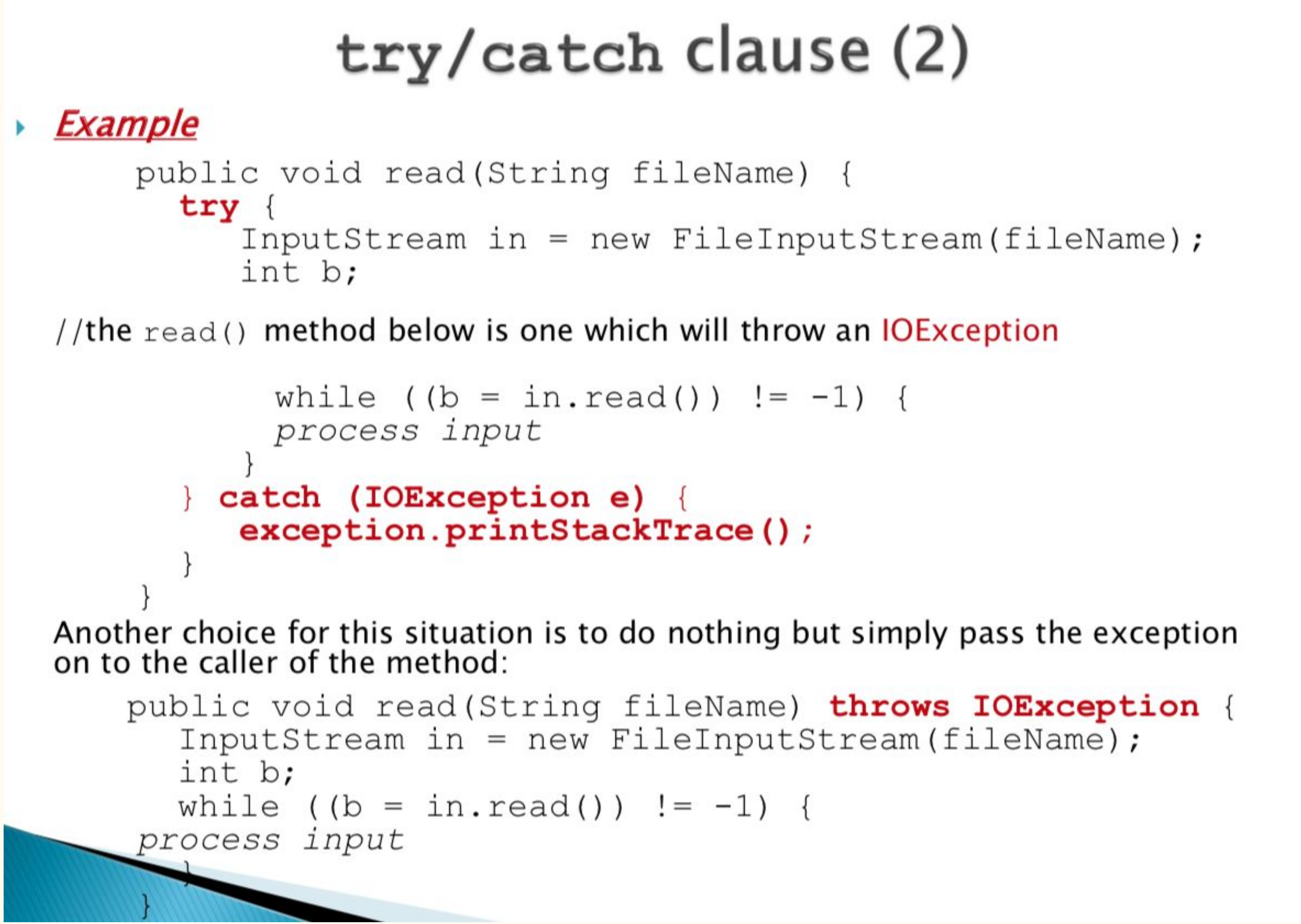
try/catch clause (1)

- Basic syntax of `try/catch` block

```
try {
   statements
} catch(exceptionType1 identifier1) {
   handler for type1
} catch(exceptionType2 identifier1) {
   handler for type2
} . . .
```

  - If no exception occurs during the execution of the statements in the **try** clause, it finishes successfully and all the **catch** clauses are skipped
  - If any of the code inside the **try** block throws an exception, either directly via a **throw** or indirectly:

    1. The program skips the remainder of the code in the **try** block
    2. The catch clauses are examined one by one, to see whether the type of the thrown exception is compatible with the type in **catch.**
    3. If an appropriate **catch** clause is found, the code inside its body gets executed and all the remaining **catch** clauses are skipped.
    4. If no such a **catch** clause is found, then the exception is thrown into an outer **try** that might have a **catch** clause to handle it



try/catch clause (2)

- ***Example***

```
    public void read(String fileName) {
      try {
         InputStream in = new FileInputStream(fileName);
         int b;
```

//the `read()` method below is one which will throw an IOException

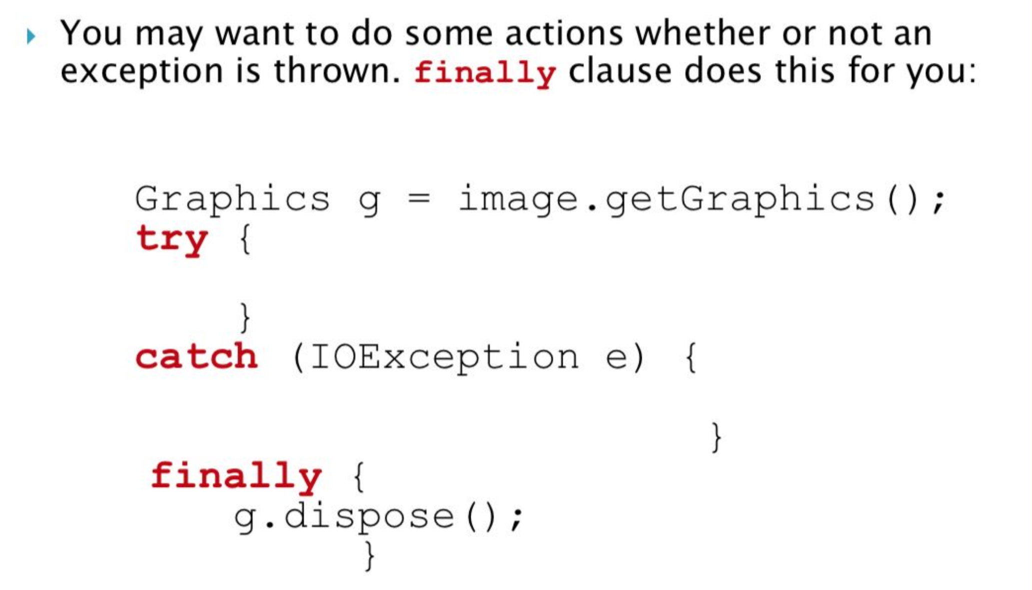
```
           while ((b = in.read()) != -1) {
           process input
          }
      } catch (IOException e) {
         exception.printStackTrace();
      }
    }
```

Another choice for this situation is to do nothing but simply pass the exception on to the caller of the method:

```
    public void read(String fileName) throws IOException {
      InputStream in = new FileInputStream(fileName);
      int b;
      while ((b = in.read()) != -1) {
    process input
      }
    }
```

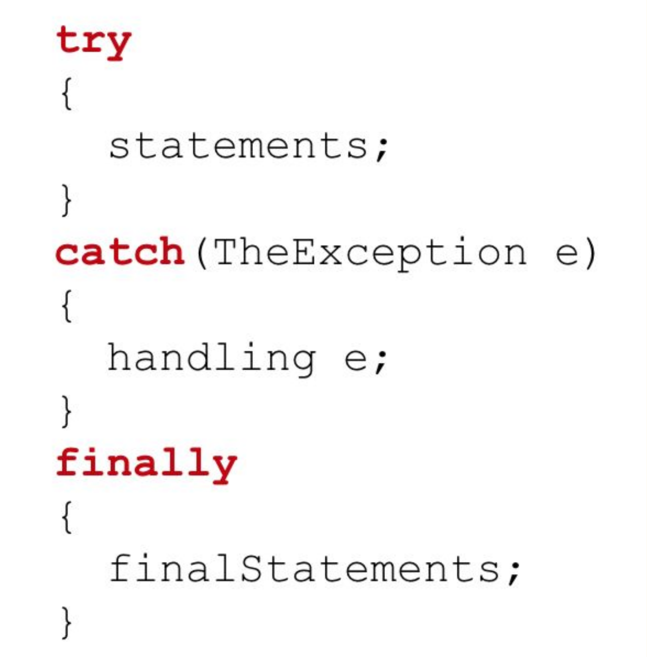
Finally

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

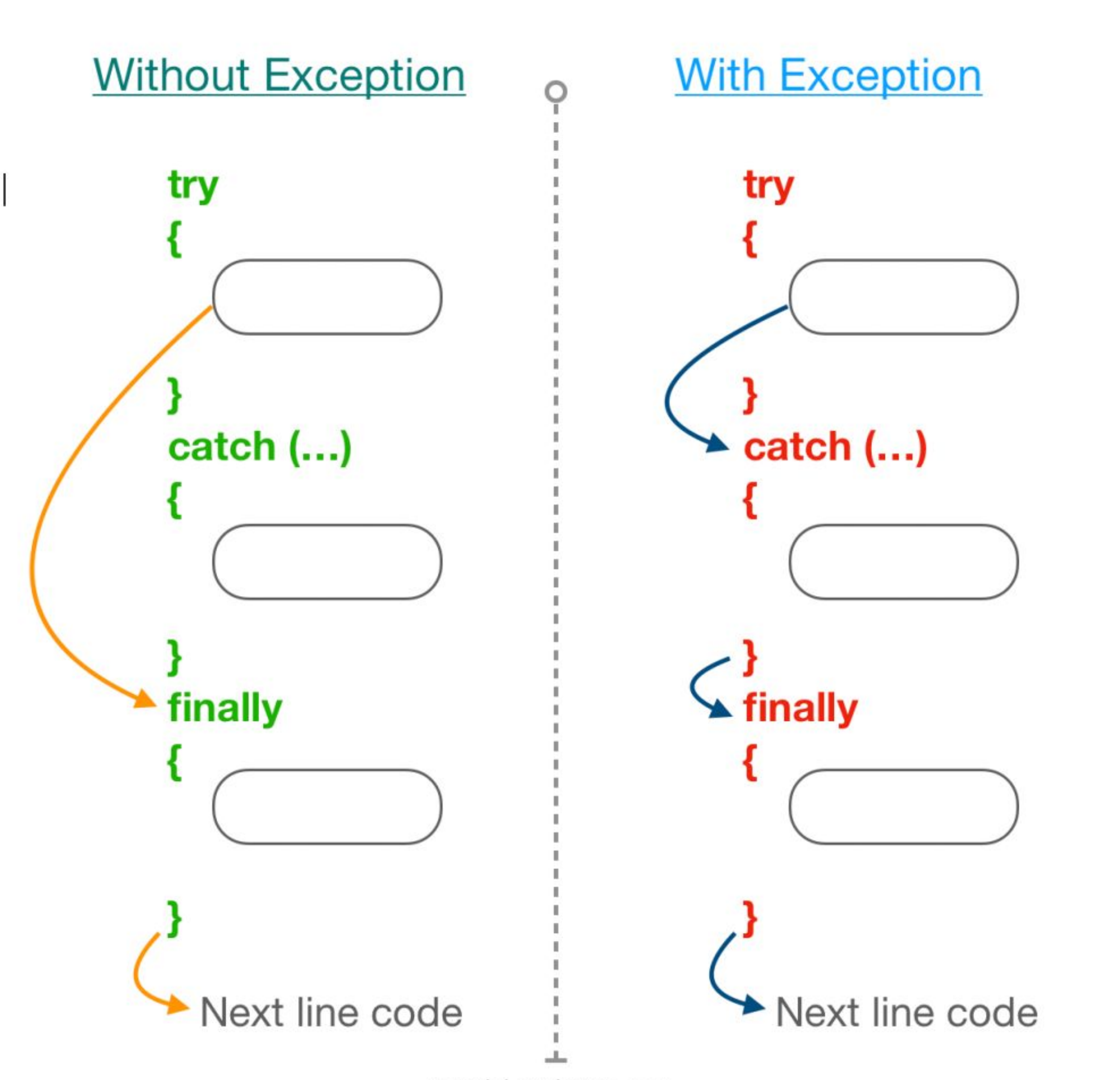


- You may want to do some actions whether or not an exception is thrown. **finally** clause does this for you:

```
Graphics g = image.getGraphics();
try {

      }
catch (IOException e) {

                                 }
 finally {
      g.dispose();
           }
```



```
try
{
  statements;
}
catch(TheException e)
{
  handling e;
}
finally
{
  finalStatements;
}
```

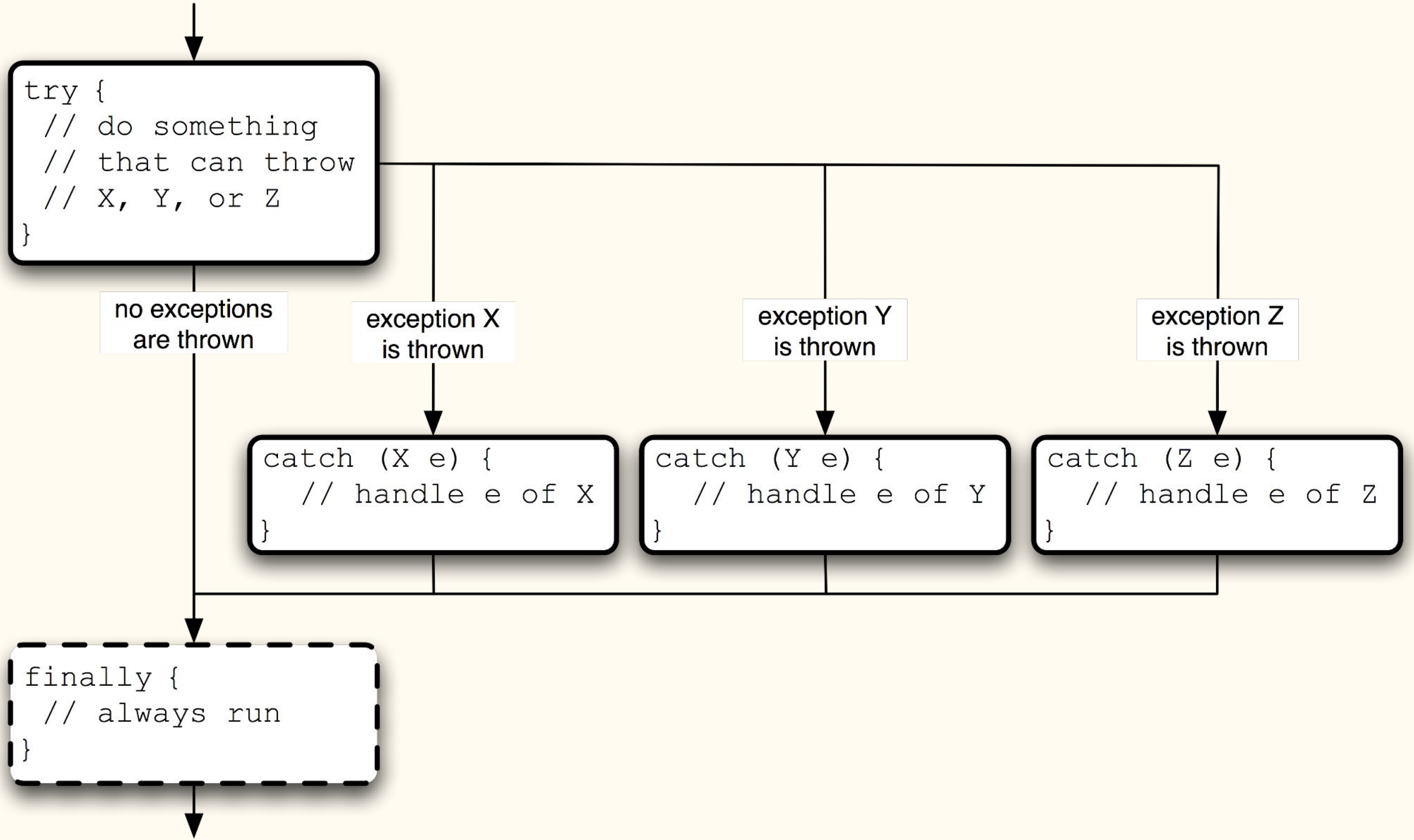


Without Exception

```
try
{

}
catch (...)
{

}
finally
{

}
Next line code
```

With Exception

```
try
{

}
catch (...)
{

}
finally
{

}
Next line code
```



```
try {
 // do something
 // that can throw
 // X, Y, or Z
}
```

no exceptions are thrown

exception X is thrown

exception Y is thrown

exception Z is thrown

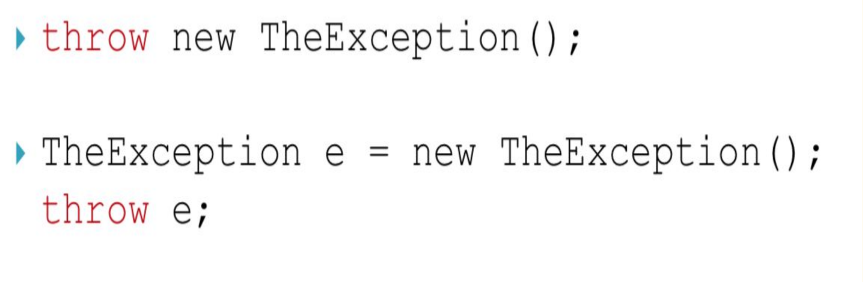
```
catch (X e) {
  // handle e of X
}
```

```
catch (Y e) {
  // handle e of Y
}
```

```
catch (Z e) {
  // handle e of Z
}
```

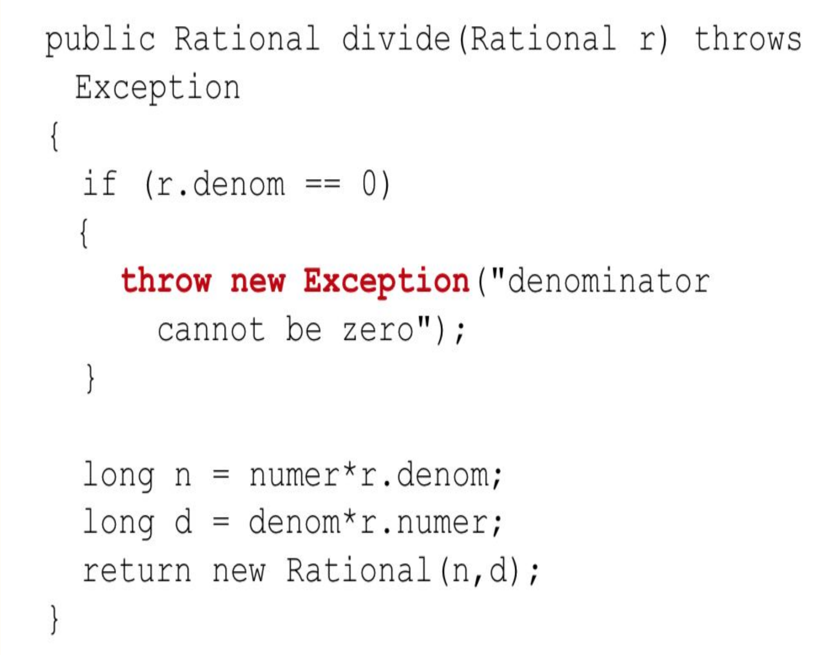
```
finally {
 // always run
}
```

Throwing Exceptions



-
  ```
  throw new TheException();
  ```

-
  ```
  TheException e = new TheException();
  throw e;
  ```

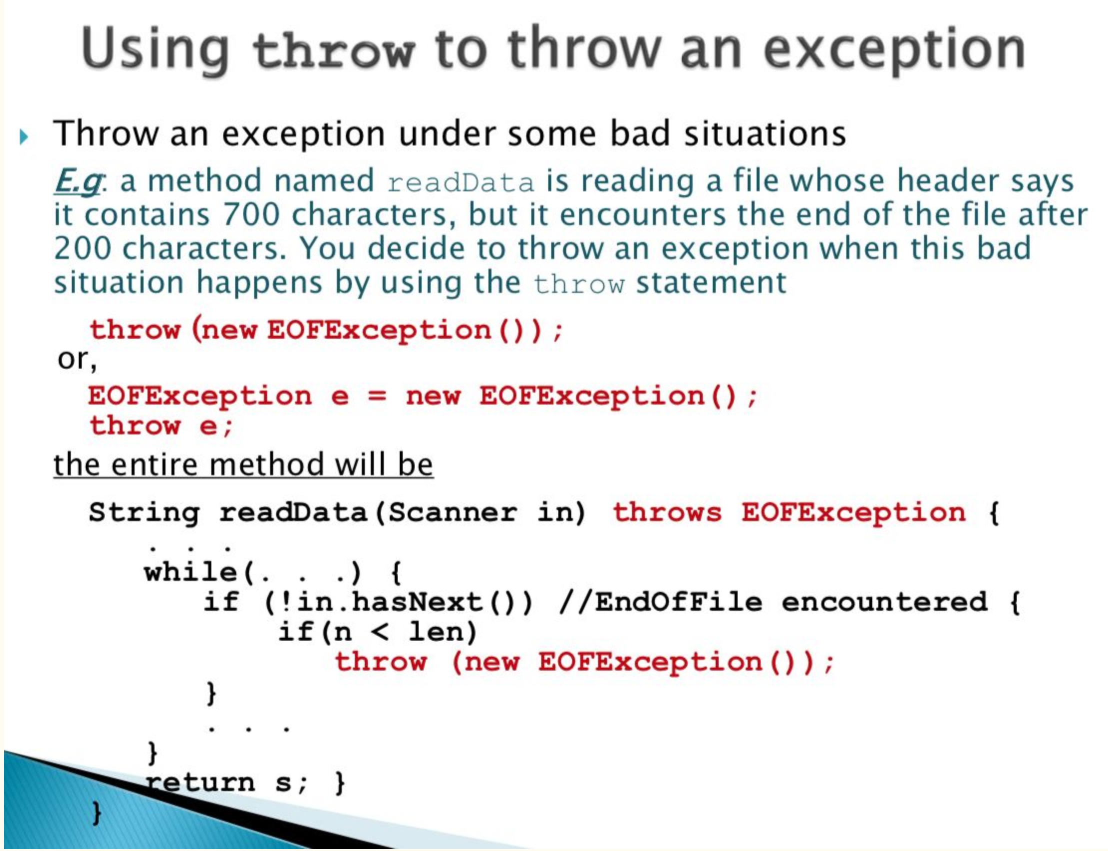


```
public Rational divide(Rational r) throws
  Exception
{
  if (r.denom == 0)
  {
    throw new Exception("denominator
      cannot be zero");
  }

  long n = numer*r.denom;
  long d = denom*r.numer;
  return new Rational(n,d);
}
```

Throwing Exceptions

- You can throw build-in or custom exception

- Exceptions can be thrown by JVM or by you. Feel your power :)



Using throw to throw an exception

- Throw an exception under some bad situations

E.g: a method named `readData` is reading a file whose header says it contains 700 characters, but it encounters the end of the file after 200 characters. You decide to throw an exception when this bad situation happens by using the `throw` statement

```
  throw (new EOFException());
```

or,

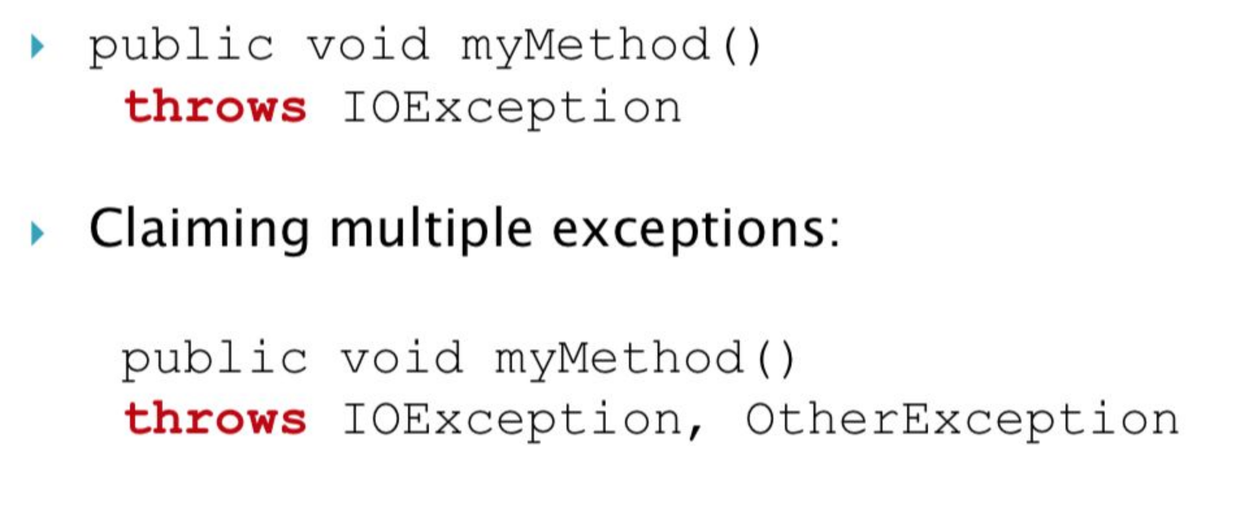
```
  EOFException e = new EOFException();
  throw e;
```

the entire method will be

```
  String readData(Scanner in) throws EOFException {
     . . .
     while(. . .) {
        if (!in.hasNext()) //EndOfFile encountered {
            if(n < len)
               throw (new EOFException());
        }
        . . .
     }
     return s; }
  }
```

Claiming Exceptions

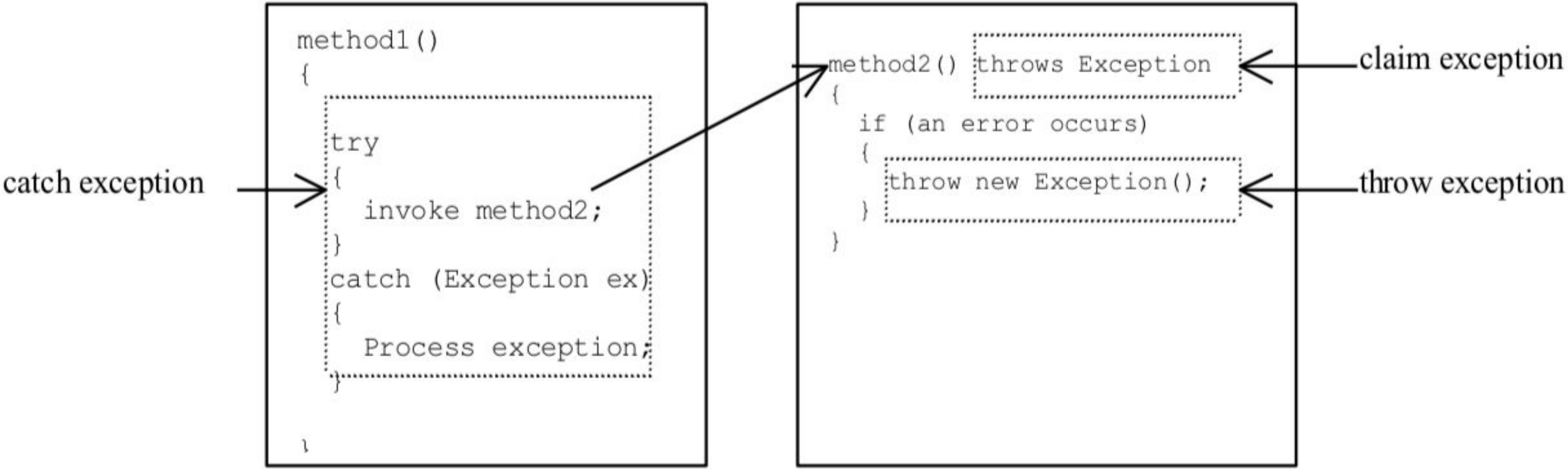
Specifying which exceptions might occur, without handling.



-
  ```
  public void myMethod()
    throws IOException
  ```
- Claiming multiple exceptions:

  ```
  public void myMethod()
  throws IOException, OtherException
  ```

Claiming, Throwing, and Catching Exceptions



catch exception

```
method1()
{
  try
  {
    invoke method2;
  }
  catch (Exception ex)
  {
    Process exception;
  }

}
```

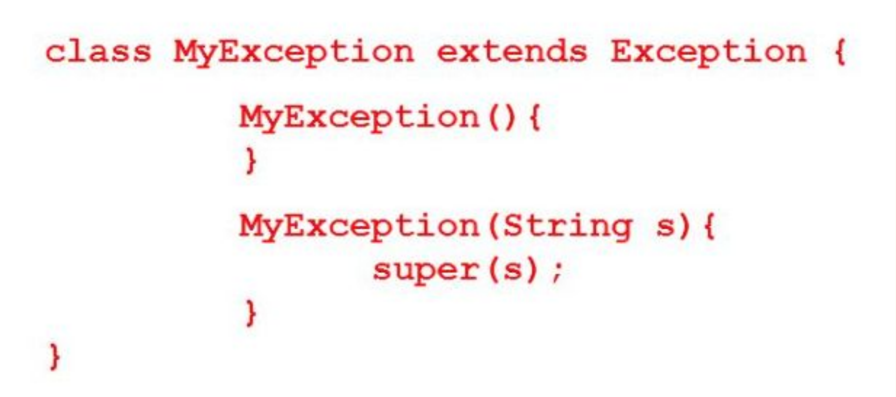
claim exception

throw exception

```
method2() throws Exception
{
  if (an error occurs)
  {
    throw new Exception();
  }
}
```

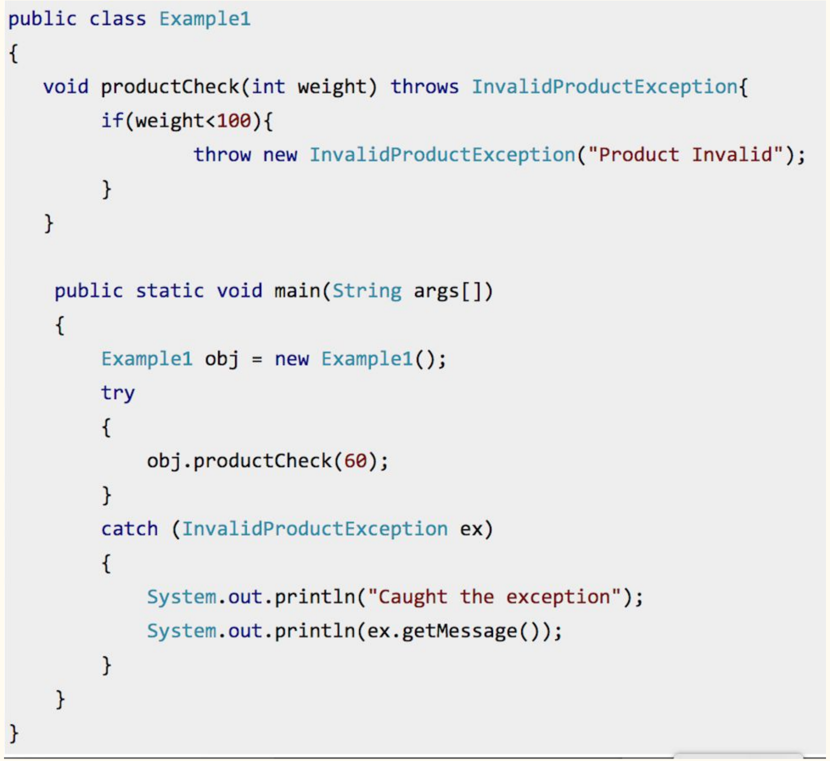
User Defined (Custom) Exceptions

- Create your own Exception by extending Exception
- Include 2 constructors:
  - default (no-arg)
  - one to allow for an error specific message

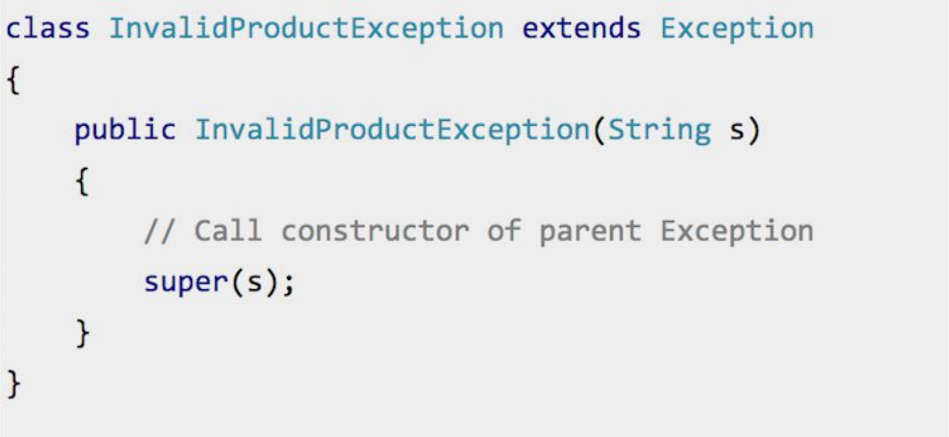


```
class MyException extends Exception {

          MyException(){
          }

          MyException(String s){
                super(s);
          }
}
```

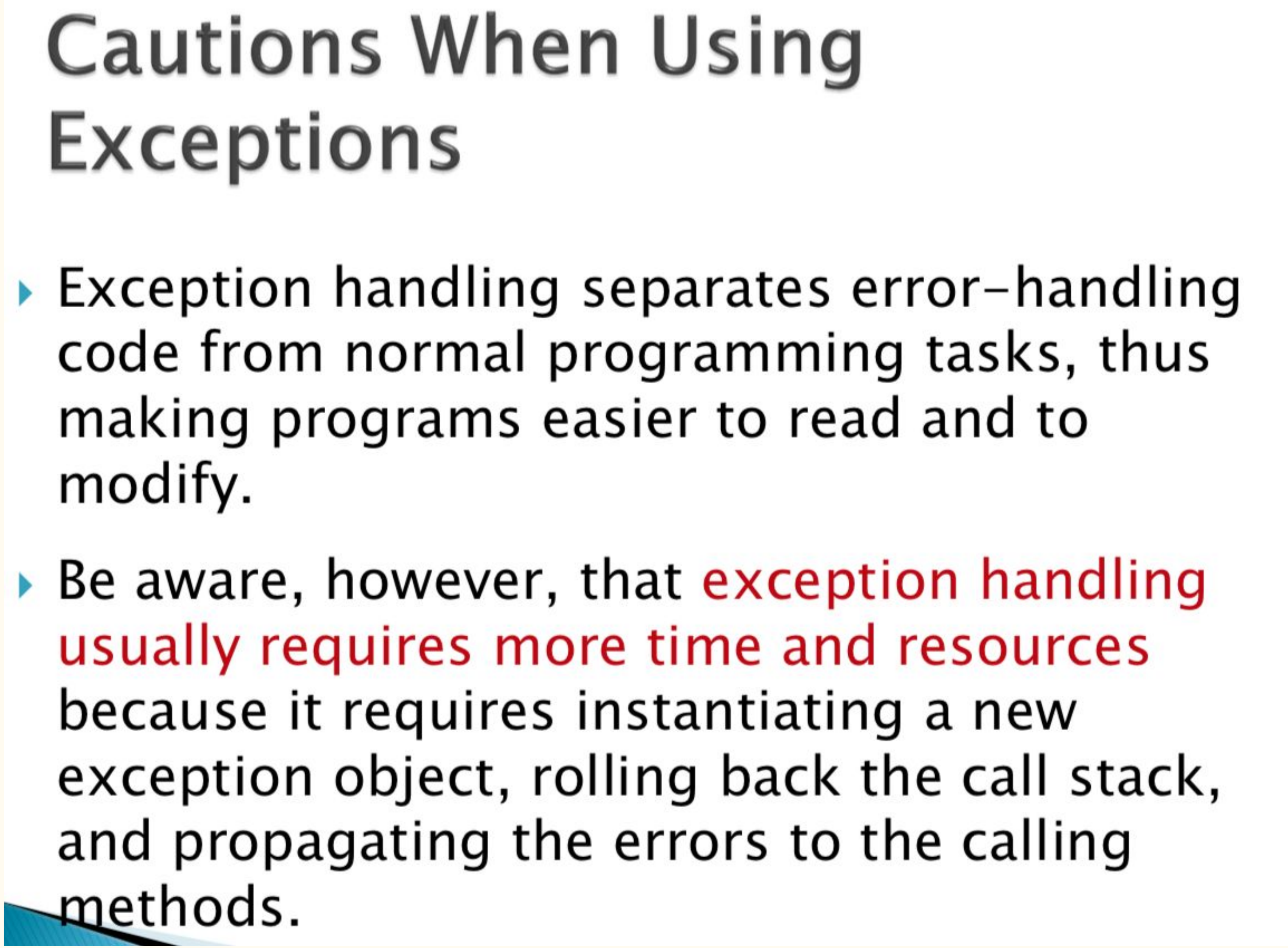
Custom Exception



```java
public class Example1
{
   void productCheck(int weight) throws InvalidProductException{
        if(weight<100){
                throw new InvalidProductException("Product Invalid");
        }
   }

    public static void main(String args[])
    {
        Example1 obj = new Example1();
        try
        {
            obj.productCheck(60);
        }
        catch (InvalidProductException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }
    }
}
```

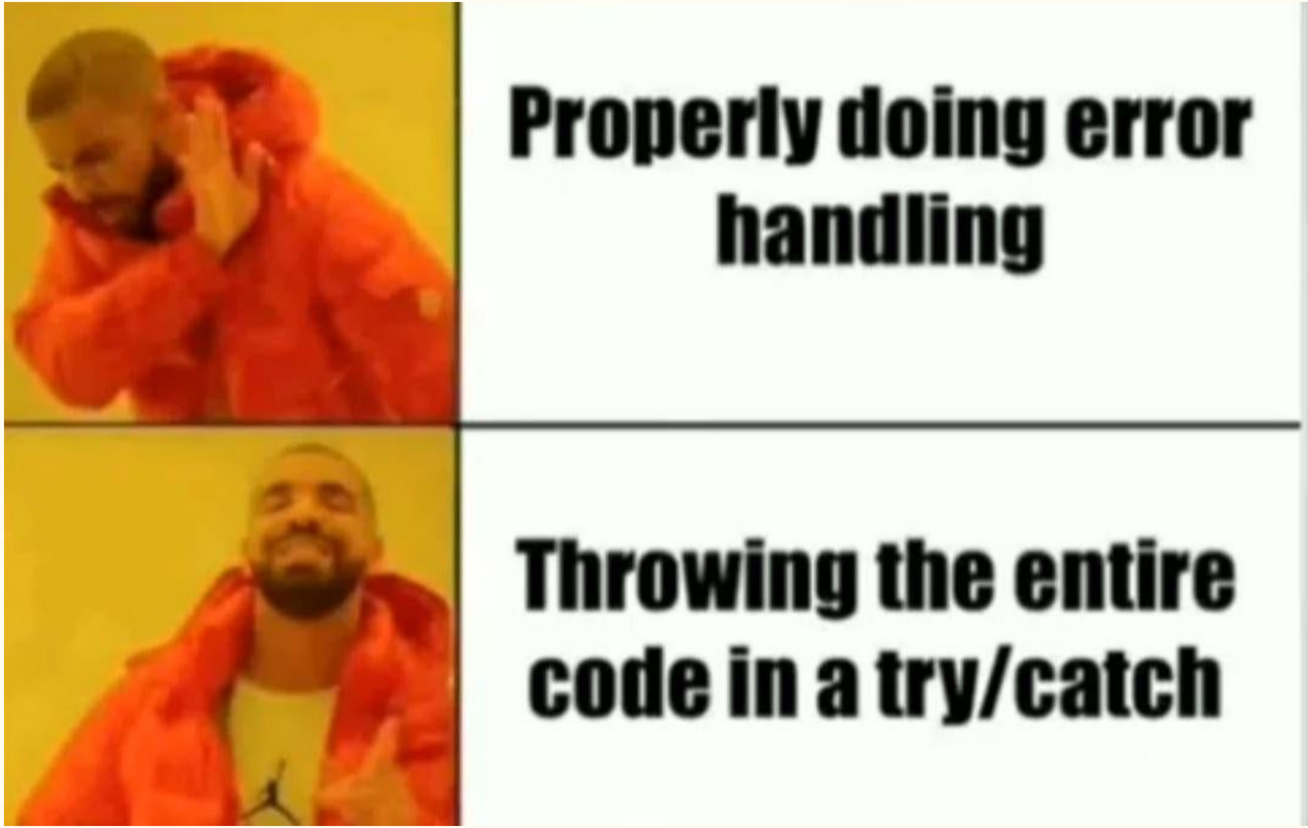


```java
class InvalidProductException extends Exception
{
    public InvalidProductException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}
```



Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.


Properly doing error handling
Throwing the entire code in a try/catch

handling all exceptions except one
client encountering that error

Try
Exception
Catch block
Unhandled Exception

About your Practice 4...

Do not be surprised to see a lot of exceptions there. This practice is about exceptions :)

YOU GET AN EXCEPTION!
AND YOU GET AN EXCEPTION!
AND YOU GET AN EXCEPTION!
EVERYONE GETS AN EXCEPTION!