OKANAGAN

COSC 121

Computer Programming II

Exception Handling

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Previous Lecture

- User defined interfaces
  - default, static, and private methods
- Java standard interfaces
  - `Comparable` interface
  - `Cloneable` interface
    - Shallow vs. deep cloning.

Outline

Today:

- Intro to Java Exceptions
- How exception handling works?
- Java Exceptions Types
  - checked vs unchecked
- How to deal with Exceptions:
  - Declaring exceptions
  - Handling exceptions

Next lecture:

- Text I/O

What are Java Exceptions?

When a program runs into a ***runtime error***, the program terminates abnormally. How can you ***handle the runtime error*** so that the program can continue to run or terminate gracefully?

Exception handling enables a program to deal with exceptional situations and continue its normal execution.

An exception is an object that represents an error or a condition that prevents execution from proceeding normally

- Examples:
  - a program tries to access an **out-of-bounds array**;
  - a program tries to connect to a website using an **invalid address**;
  - a program runs **out of memory**;
  - a program tries to read a file of integers but finds a string value in the file

If the exception is not handled, the program will terminate abnormally.

Example

```java
import java.util.Scanner;
public class ArrayReading {
    public static void main(String[] args) {
        int[] numbers = {5, 7, 3};
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the index: ");
        int i = input.nextInt();
        printElement(numbers, i);
        input.close();
    }
    private static void printElement(int[] numbers, int i) {
        System.out.println(numbers[i]);
    }
}
```

Output

```
Enter the index: 1
7
```

```
Enter the index: 5
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
	at ch12.ArrayReading.printElement(ArrayReading.java:14)
	at ch12.ArrayReading.main(ArrayReading.java:10)
```

Example, cont.

Fix #1: **Defensive Programming**

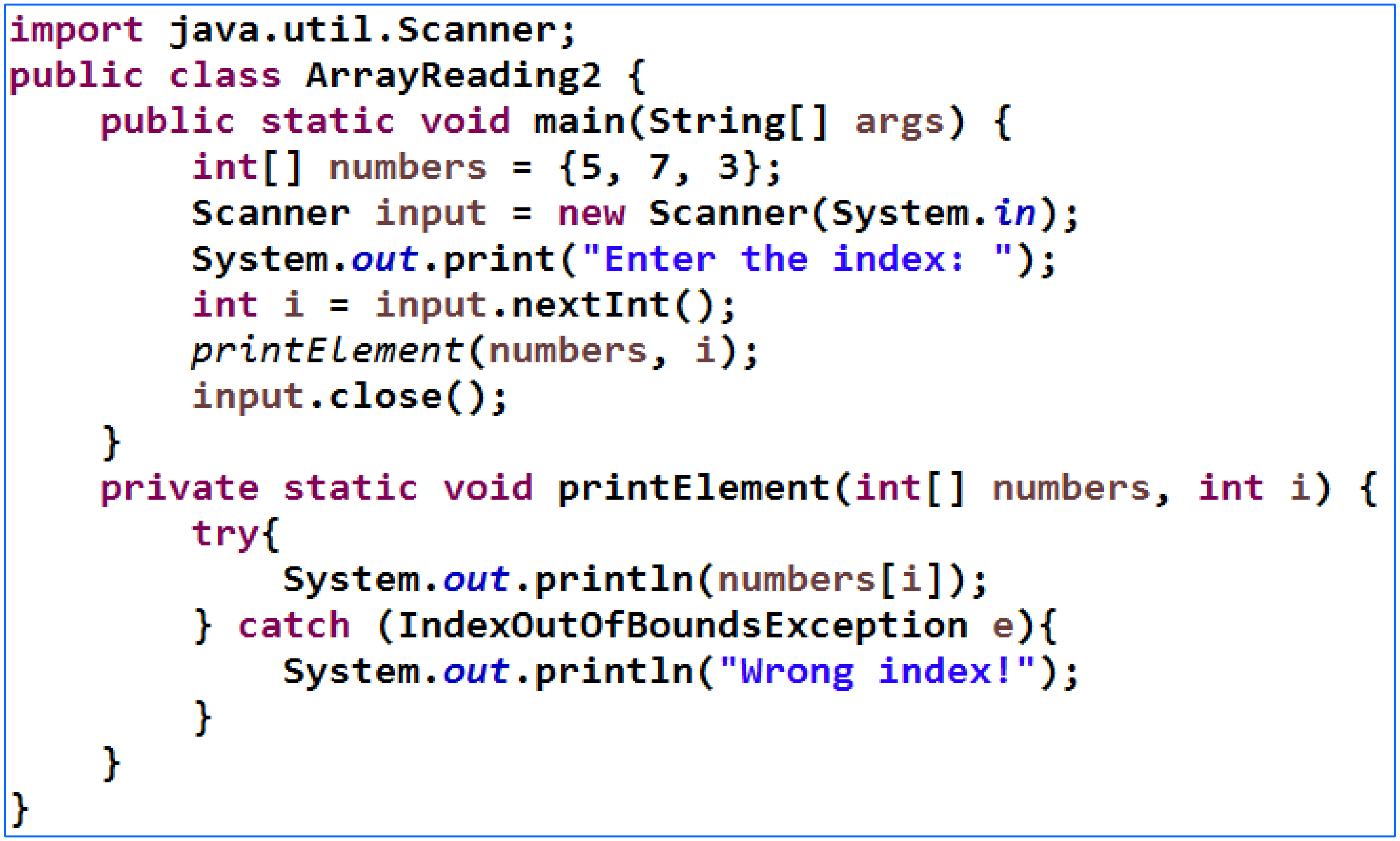
Use `if` statement to check the index value.

```java
import java.util.Scanner;
public class ArrayReading {
    public static void main(String[] args) {
        int[] numbers = {5, 7, 3};
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the index: ");
        int i = input.nextInt();
        printElement(numbers, i);
        input.close();
    }
    private static void printElement(int[] numbers, int i) {
        if(i >= numbers.length || i < 0)
            System.out.println("Wrong index!");
        else
            System.out.println(numbers[i]);
    }
}
```

Example, cont.

Fix #2: **Exception Handling**

Use `try..catch` statement to handle exceptions.



```java
import java.util.Scanner;
public class ArrayReading2 {
    public static void main(String[] args) {
        int[] numbers = {5, 7, 3};
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the index: ");
        int i = input.nextInt();
        printElement(numbers, i);
        input.close();
    }
    private static void printElement(int[] numbers, int i) {
        try{
            System.out.println(numbers[i]);
        } catch (IndexOutOfBoundsException e){
            System.out.println("Wrong index!");
        }
    }
}
```

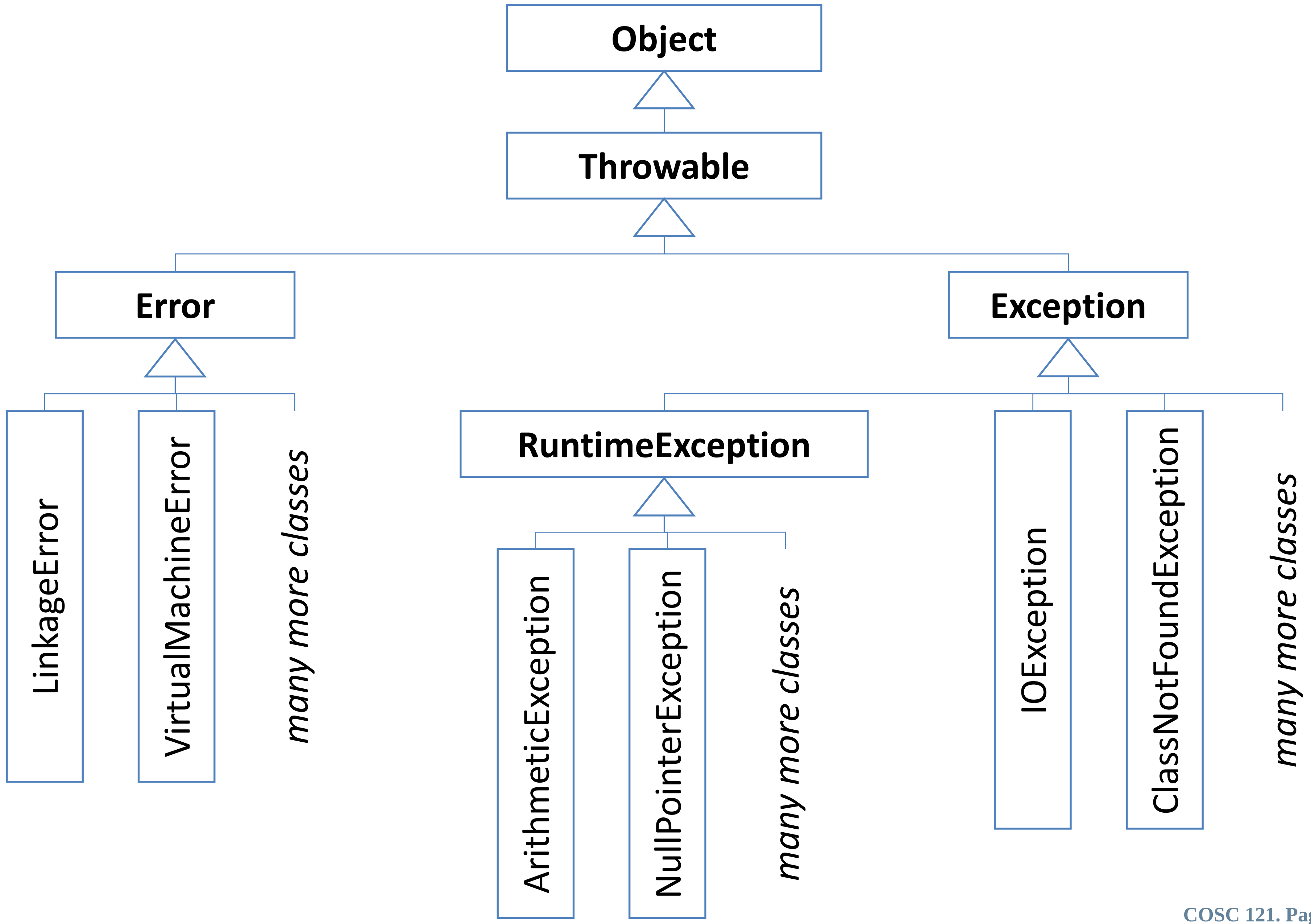
More about this later!

How exception handling works?

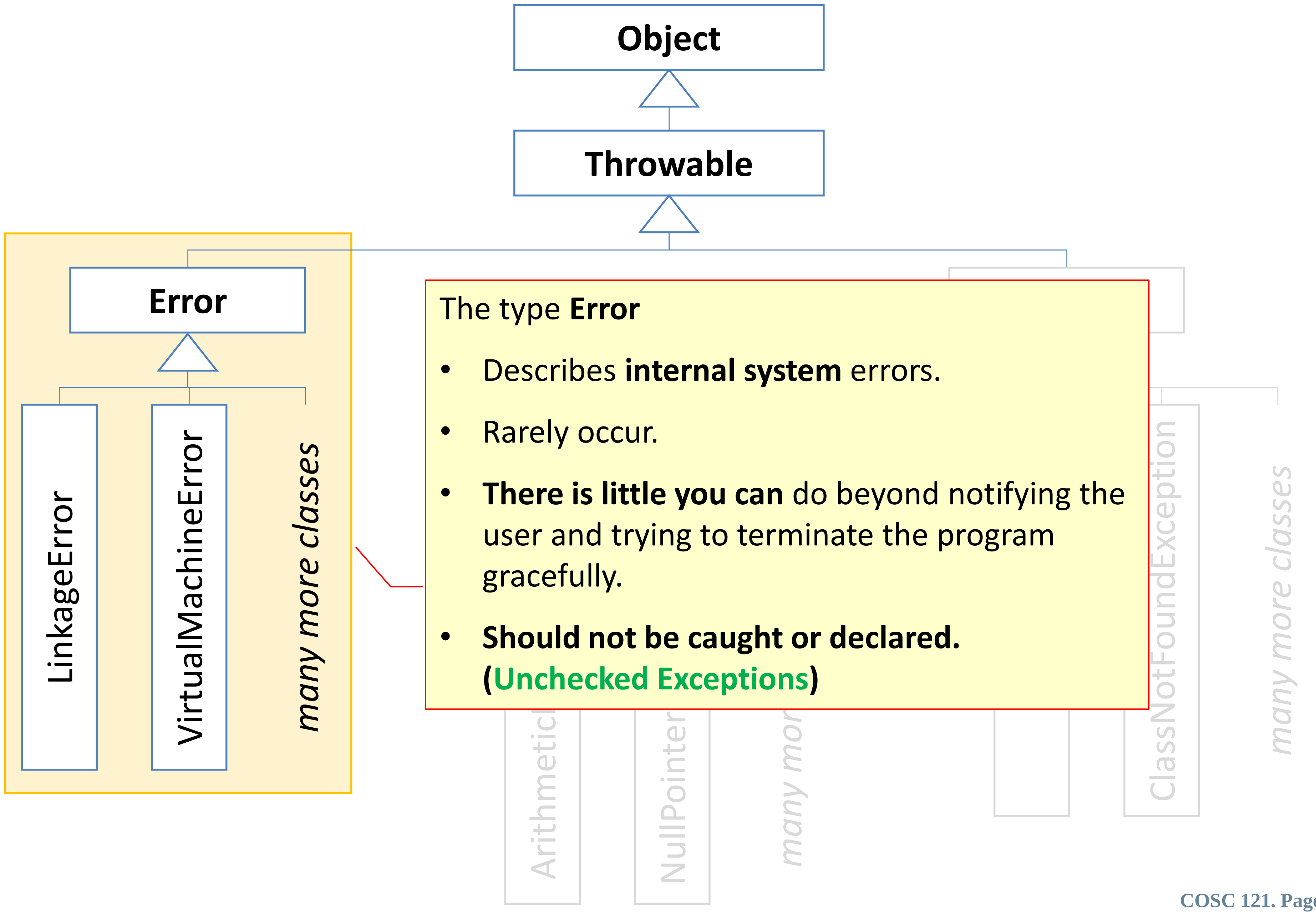
1. When an **error** is detected, Java **stops the normal flow** of program execution.
   - *the code **throws** an **exception**.*
2. An "**exception object**" is created.
   - This object stores information about the error.
   - There are different kinds of exception classes.
3. Java transfers control from the part of the program where the error occurred to an exception handler, which deals with the situation.
4. The exception object is passed to the exception handler code where you can use it to decide on the appropriate action.

Note that: Exceptions are objects, and objects are defined using classes. The root class for exceptions is **java.lang.Throwable**

Java Exceptions



Java Exceptions



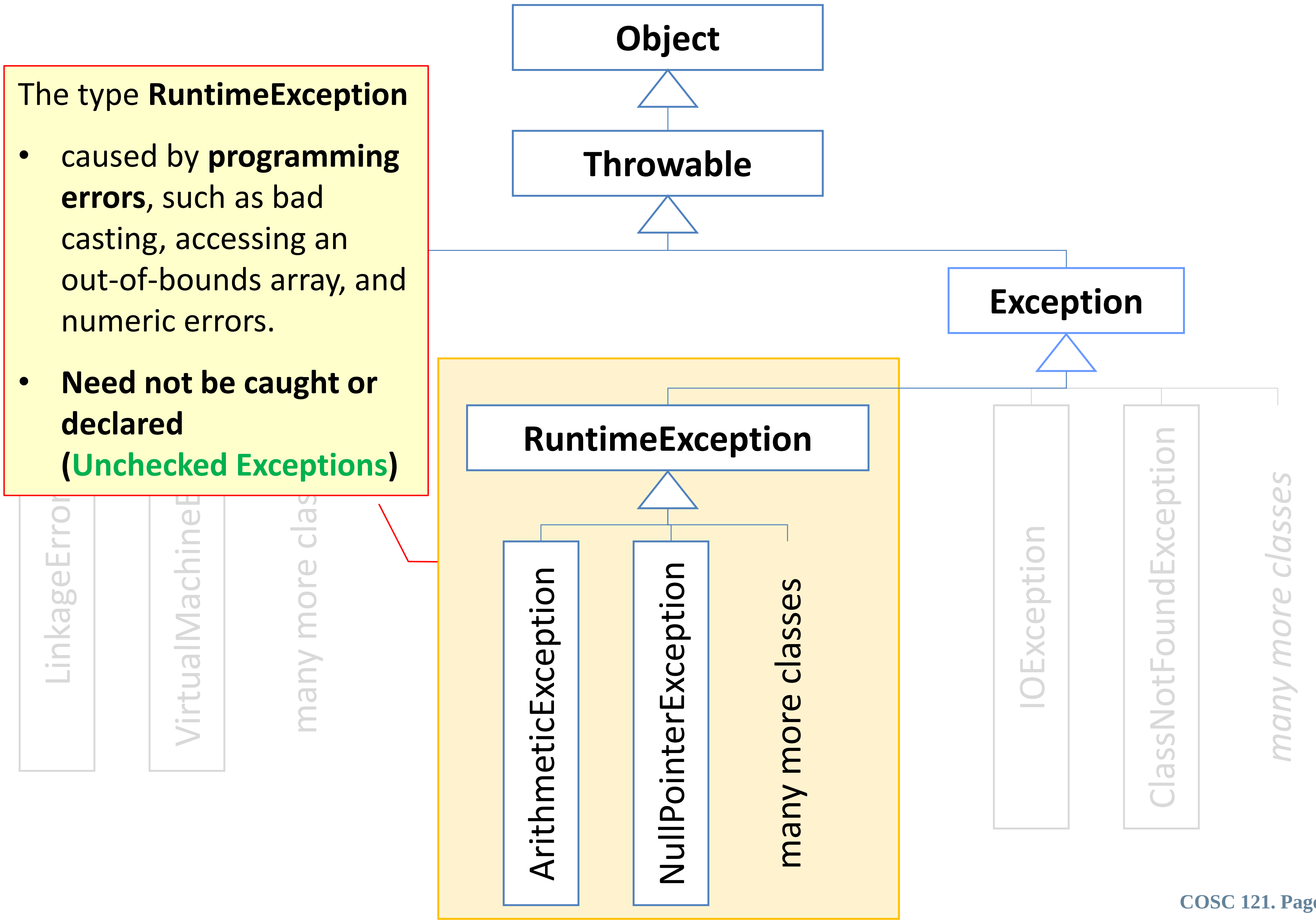
The type **Error**

- Describes **internal system** errors.
- Rarely occur.
- **There is little you can** do beyond notifying the user and trying to terminate the program gracefully.
- **Should not be caught or declared. (Unchecked Exceptions)**

Java Exceptions

The type **RuntimeException**

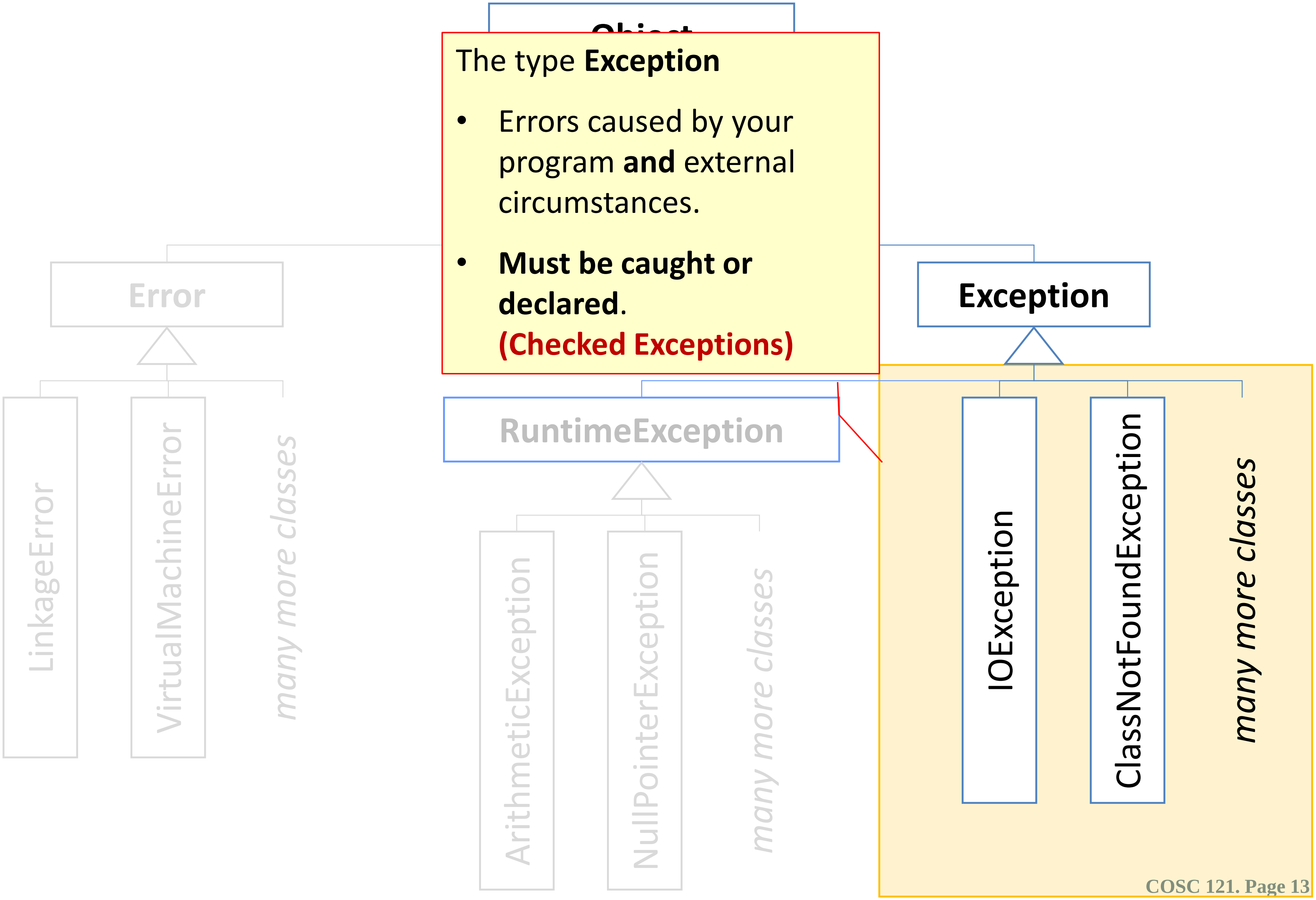
- caused by **programming errors**, such as bad casting, accessing an out-of-bounds array, and numeric errors.
- **Need not be caught or declared (Unchecked Exceptions)**



Java Exceptions

The type **Exception**

- Errors caused by your program **and** external circumstances.
- **Must be caught or declared.**
  (Checked Exceptions)



Unchecked Exceptions

`RuntimeException`, **`Error`** and their subclasses.

They can occur anywhere in a program, usually because of programming logic errors

- To avoid cumbersome overuse of try-catch blocks, Java **does not require you** to write code to handle them.

Examples

- Subclasses of **`Error`**
  - **`VirtualMachineError`** the **JVM is broken** or has run out ofa the resources it needs in order to continue operating
- Subclasses of **`RunTimeException`**:
  - **`ArithmeticException:`** Dividing an integer **by zero.**
  - **`IndexOutOfBoundsException:`** accessing an **index** (e.g., of an array) outside the valid limits.
  - **`NumberFormatException:`** when you try to **convert a String to a numeric value**, but that the string does not have the appropriate format.
    - e.g., int x = Integer.parseInt(s); //error if e.g., s = “abc”
  - **`NullPointerException:`** accessing an object through a reference variable before an object is assigned to it.

Checked Exceptions

`Exception` subclasses **<u>except</u>** **`RuntimeException`**.

The compiler **forces** you to check and deal with them.

- No default action when encountered (i.e. Java doesn't throw nor catch the exception.. Needs you to tell it what to do).

Examples:

- Subclasses of **`IOException`**
  - **`EOFException`** occurs when a program attempts to read past the end of a file.
  - **`FileNotFoundException`** can occur when a program attempts to open or write to an existing file, but the file is not found.
  - **`MalformedURLException`** indicates that an invalid form of URL (such as a website address) has occurred.

Remember: How exception handling works?

1. When an **error** is detected, Java **stops the normal flow** of program execution.

2. An “exception object” is created.

3. Java transfers control from the part of the program where the error occurred to an exception handler.

4. The exception object is passed to the exception handler code where ***you can use it to decide on the appropriate action.***

How to Deal with Exceptions?

For ***checked exception***, you must do one of two things (you can also use them with unchecked exceptions, but you don't have to):

(1) Declaring exceptions: Declare in the method header that an exception may be thrown. In this case **you do not handle the exception within the method**, but pass it on to the calling method.

(2) Handling exceptions: catch the exception and deal with it within the method, using a **try-catch** statement.

For ***unchecked exceptions*****,** Java uses **technique (1) by default** (unless you write code to change that)

(1) Declaring exceptions

Declaring an exception causes the methods where the exception occurs to throw the exception to the *calling method*.

- To declare an exception in a method, use the keyword **`throws`** followed by the **type of exception** in the method header.

```
public void m1() throws IOException
```

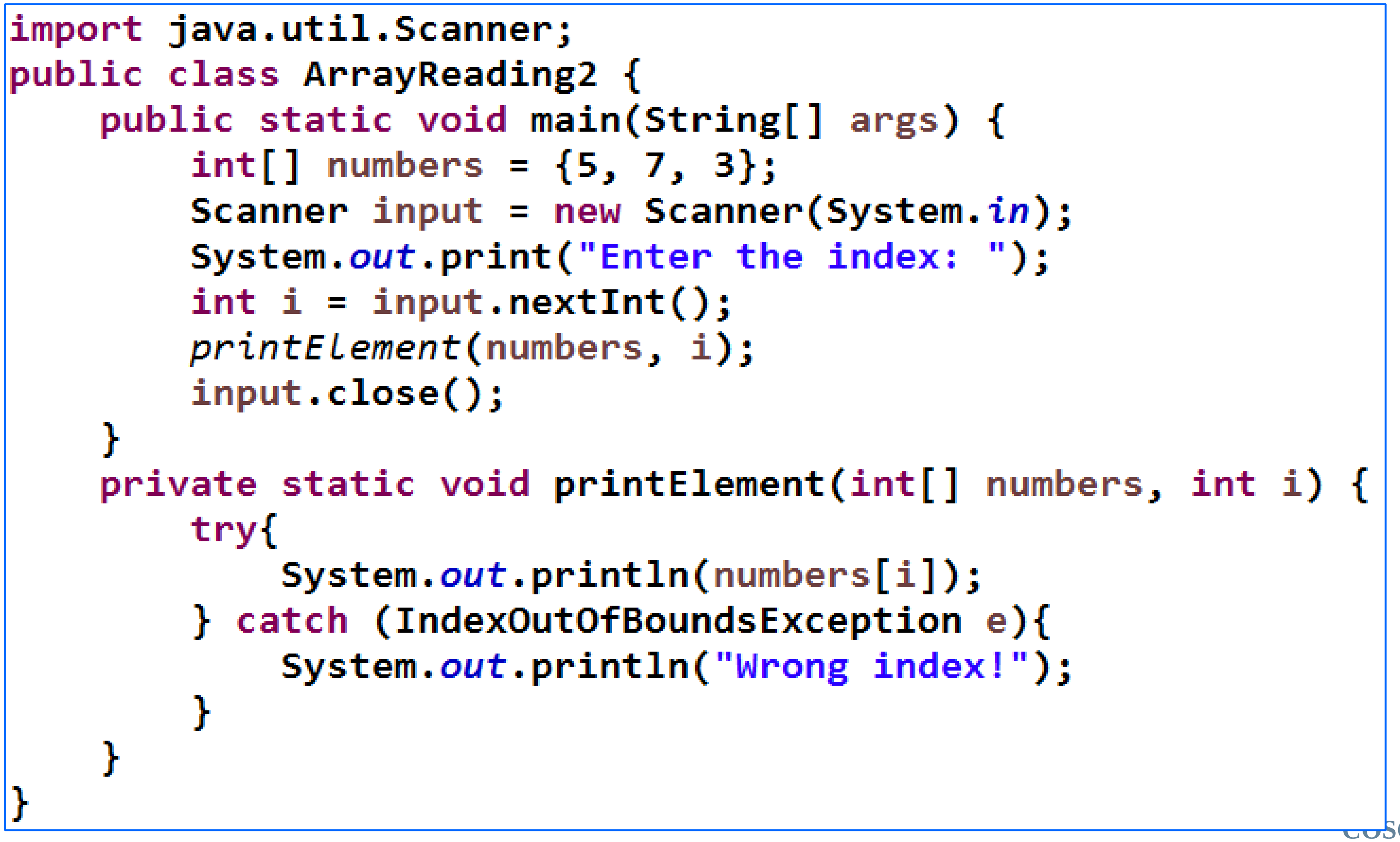
- If the method might throw multiple exceptions, use the following notation

```
public void m1() throws Exception1,...,ExceptionN
```

(2) Handling exceptions

To deal with an exception within a method, use **try-catch**

Example: When this handler code is executed, the variable within the catch clause will contain a reference to the exception object that has been thrown

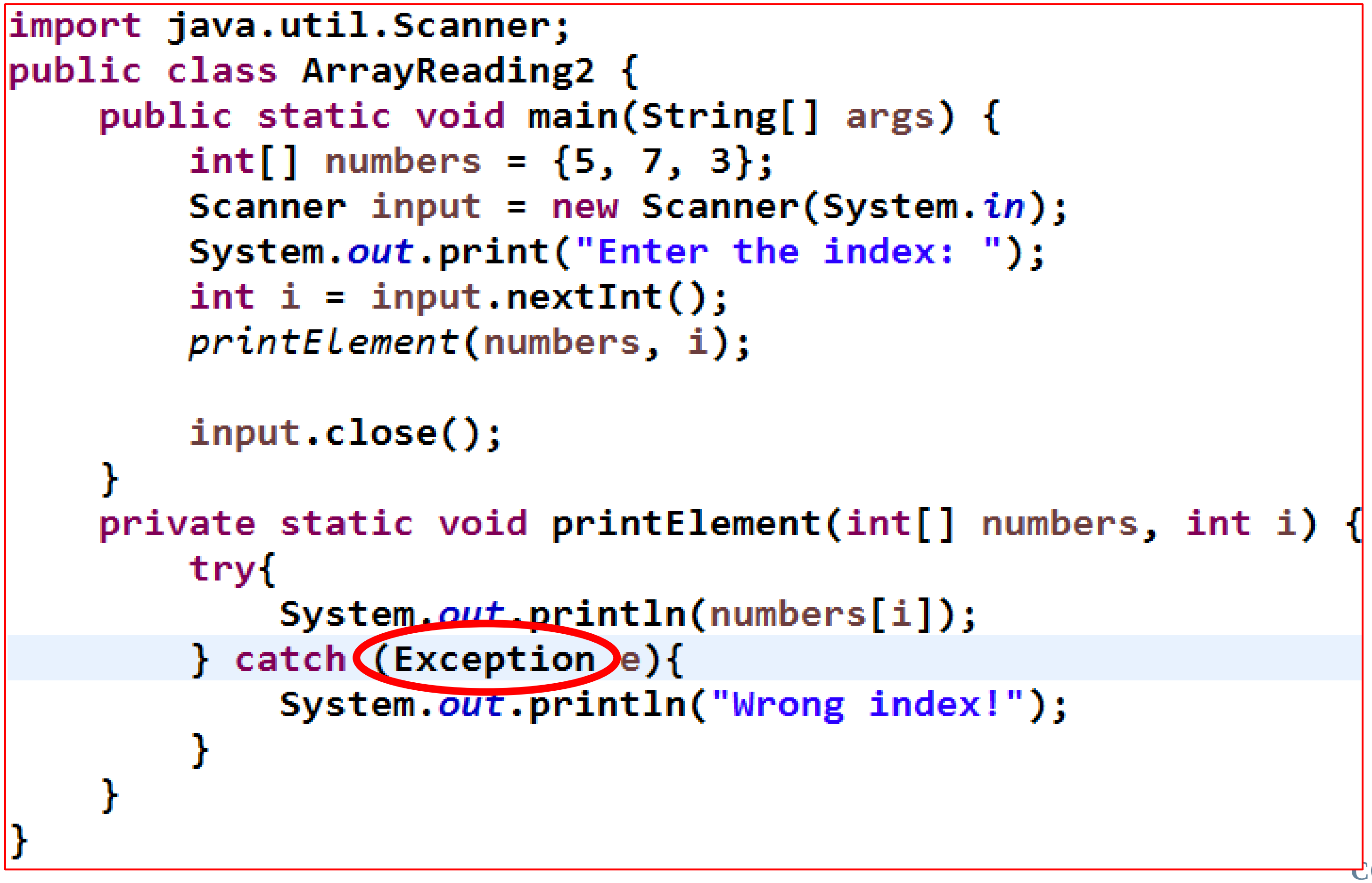


```java
import java.util.Scanner;
public class ArrayReading2 {
    public static void main(String[] args) {
        int[] numbers = {5, 7, 3};
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the index: ");
        int i = input.nextInt();
        printElement(numbers, i);
        input.close();
    }
    private static void printElement(int[] numbers, int i) {
        try{
            System.out.println(numbers[i]);
        } catch (IndexOutOfBoundsException e){
            System.out.println("Wrong index!");
        }
    }
}
```

(2) Handling exceptions, cont.

You can also catch an exception using a parent class of the thrown exception.

- Why is this possible?



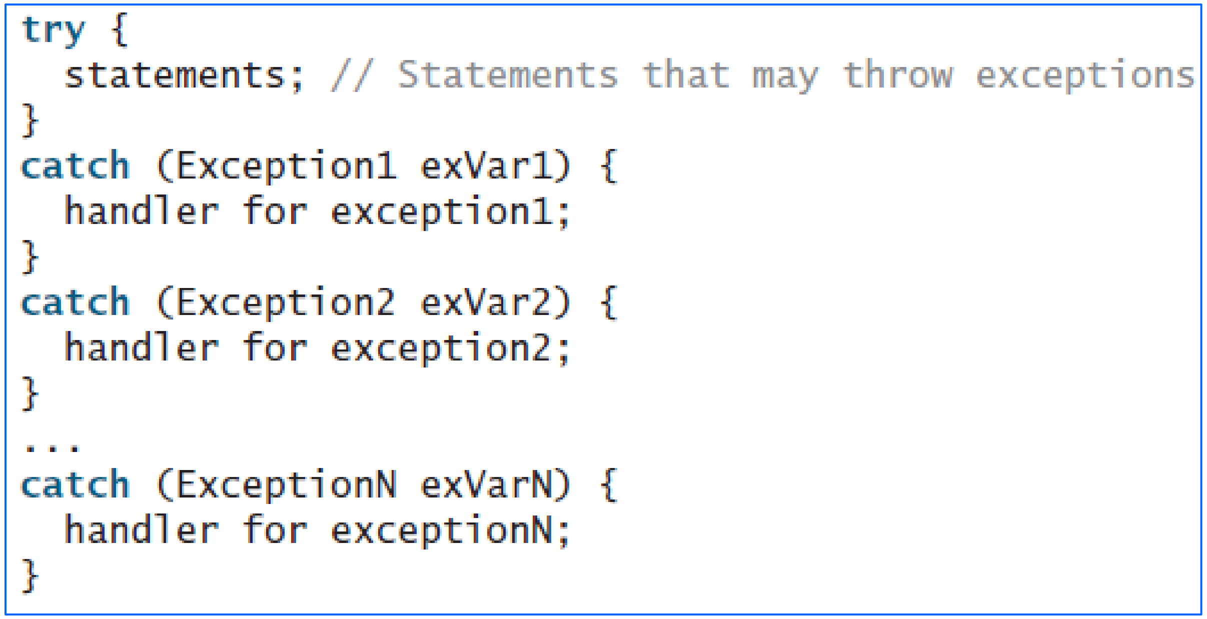
```
import java.util.Scanner;
public class ArrayReading2 {
    public static void main(String[] args) {
        int[] numbers = {5, 7, 3};
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the index: ");
        int i = input.nextInt();
        printElement(numbers, i);

        input.close();
    }
    private static void printElement(int[] numbers, int i) {
        try{
            System.out.println(numbers[i]);
        } catch (Exception e){
            System.out.println("Wrong index!");
        }
    }
}
```

.

(2) Handling exceptions, cont.

If more than one type of exception is possible, we can add additional catch clauses, like this:



```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

Each catch block is **examined in turn**, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch block.

(2) Handling exceptions, cont.

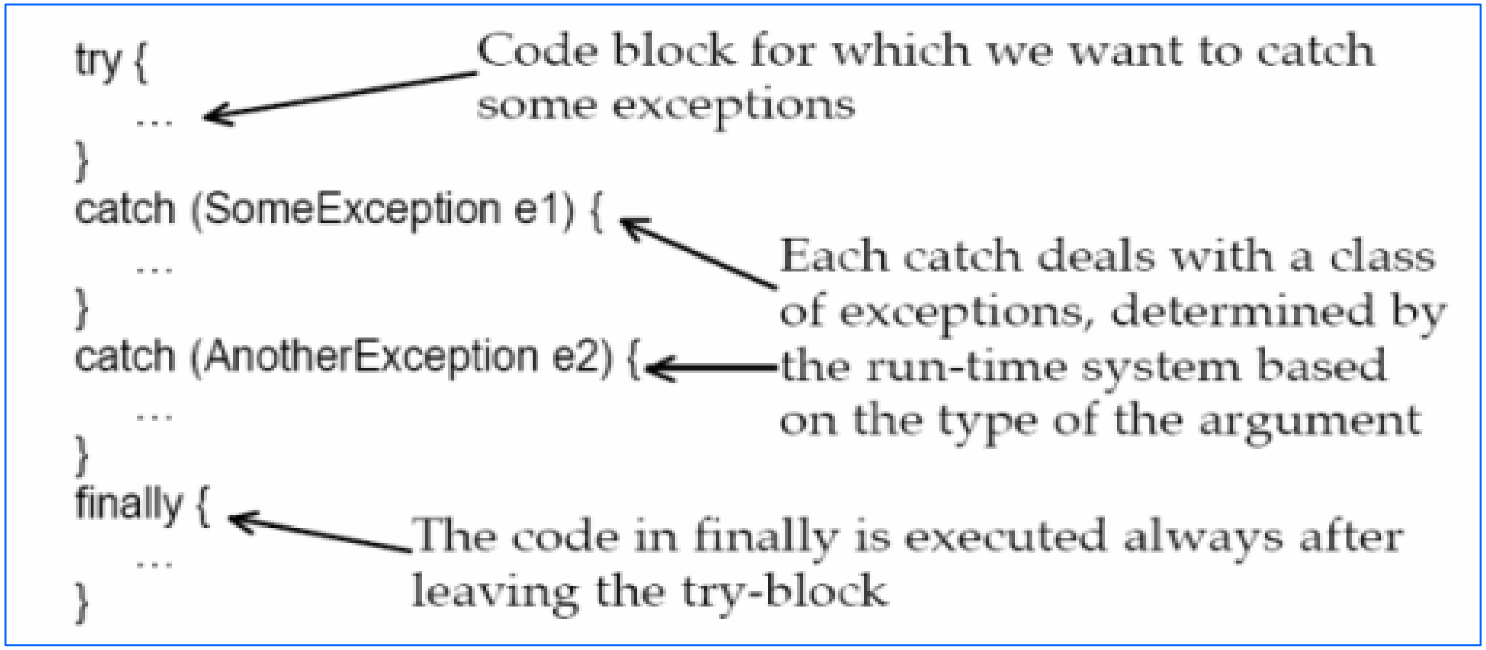
It is possible to have more than one try statement within a method. This clearly defines the area of code where each exception is expected to arise:

```
try
{
    // code that may throw a FileNotFoundException
}
catch (FileNotFoundException ex)
{
    // code that handles a FileNotFoundException
}
//
// other code
//
try
{
    // code that may throw an EOFException
}
catch (EOFException ex)
{
    // code that handles an EOFException
}
```

(2) Handling exceptions, cont.

`finally` clause

The **finally** clause is always executed regardless whether an exception occurred or not. It may be used to 'clean up' by releasing any resources such as memory or files that a method has been using before the exception was thrown.



```
try {
    ...
}
catch (SomeException e1) {
    ...
}
catch (AnotherException e2) {
    ...
}
finally {
    ...
}
```

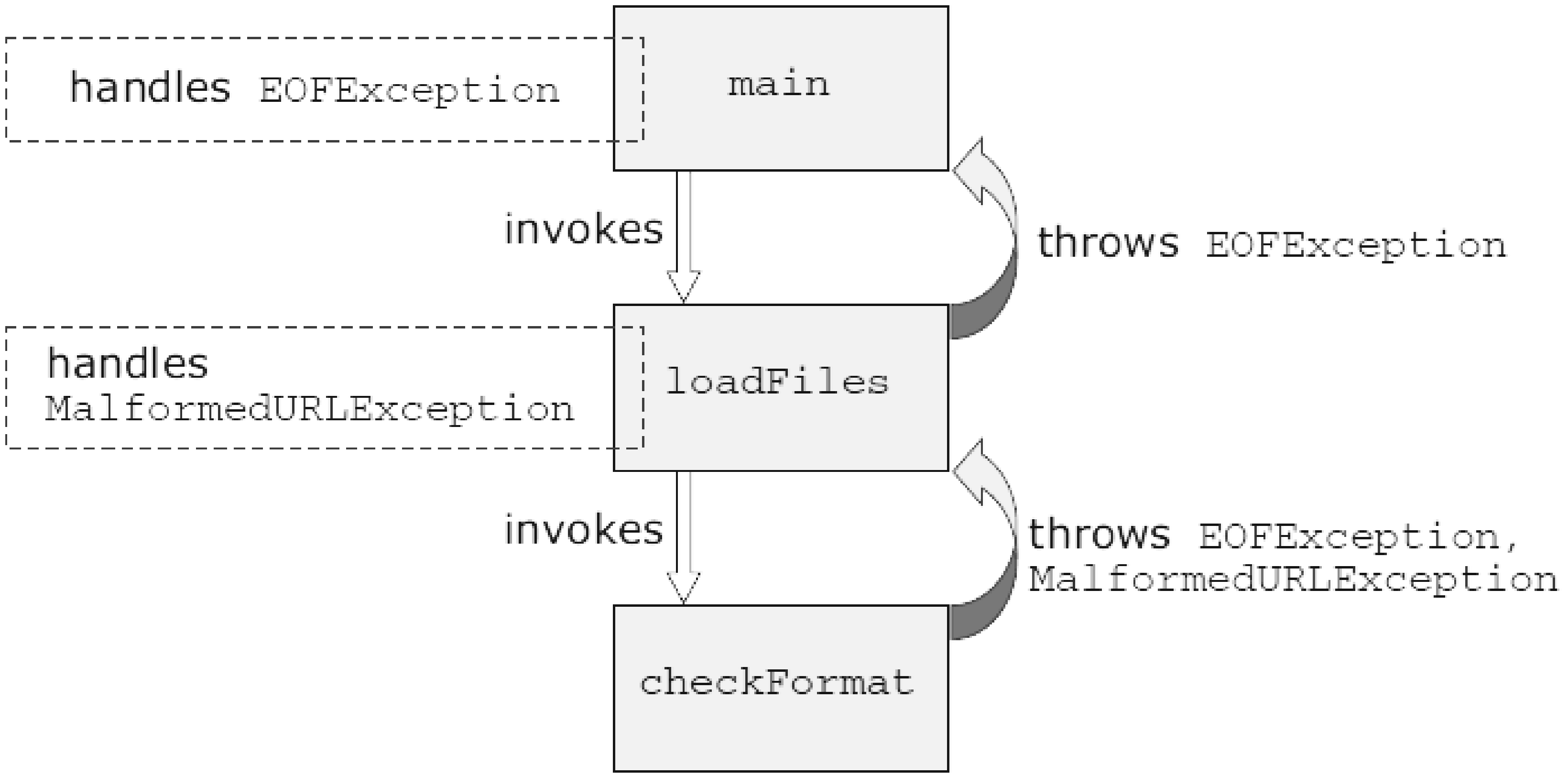
Code block for which we want to catch some exceptions
Each catch deals with a class of exceptions, determined by the run-time system based on the type of the argument
The code in finally is executed always after leaving the try-block

Source: http://developersbook.com/corejava/interview-questions/try-catch.png

Exception propagation

If no handler is found in a method, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler.

If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.



Example 1: using try-catch

Can you explain the order of execution?

```
public class Ex1 {

    public static void main(String[] args) {
        System.out.println("Start of main method");
        divide(5,0);
        System.out.println("End of main method");
    }

    public static void divide(int n1, int n2) {
        System.out.println("Start of divide method");
        try {
            System.out.println("Begin of try");
            System.out.println(n1/n2);
            System.out.println("End of try");
        }catch(ArithmeticException e) {
            System.out.println("Arith Error.");
        }
        System.out.println("End of divide method");
    }
}
```

OUTPUT

```
Start of main method
Start of divide method
Begin of try
Arith Error.
End of divide method
End of main method
```

Example 2: no try-catch

Can you explain the order of execution?

```
public class Ex1 {

    public static void main(String[] args) {
        System.out.println("Start of main method");
        divide(5,0);
        System.out.println("End of main method");
    }

    public static void divide(int n1, int n2) {
        System.out.println("Start of divide method");
        System.out.println(n1/n2);
        System.out.println("End of divide method");
    }
}
```

OUTPUT

```
Start of main method
Start of divide method
Exception in thread...
```

Example 3: Using a Parent Exception Class

Can the try-catch block catch the ArithmeticException? Explain.

```
public class Ex1 {

    public static void main(String[] args) {
        System.out.println("Start of main method");
        divide(5,0);
        System.out.println("End of main method");
    }

    public static void divide(int n1, int n2) {
        System.out.println("Start of divide method");
        try {
            System.out.println("Begin of try");
            System.out.println(n1/n2);
            System.out.println("End of try");
        }catch(Exception e) {
            System.out.println("Error");
        }
        System.out.println("End of divide method");
    }
}
```

OUTPUT

```
Start of main method
Start of divide method
Begin of try
Error
End of divide method
End of main method
```

Example 4: Catch with a different type

Why is the exception found in the “main” method (see OUTPUT)?

```
public class Ex1 {

    public static void main(String[] args) {
        System.out.println("Start of main method");
        divide(5,0);
        System.out.println("End of main method");
    }

    public static void divide(int n1, int n2) {
        System.out.println("Start of divide method");
        try {
            System.out.println("Begin of try");
            System.out.println(n1/n2);
            System.out.println("End of try");
        }catch(NullPointerException e) {
            System.out.println("Null Pointer Error");
        }
        System.out.println("End of divide method");
    }
}
```

OUTPUT

```
Start of main method
Start of divide method
Begin of try
Exception in thread "main"...
```

Example 5: Several Catch Statements

Can you explain the flow of execution?

```
public class Ex1 {
    public static void main(String[] args) {
        System.out.println("Start of main method");
        try {
            divide(5,0);
        } catch(ArithmeticException e) {
            System.out.println("Arithmetic Error");
        }
        System.out.println("End of main method");
    }
    public static void divide(int n1, int n2) {
        System.out.println("Start of divide method");
        try {
            System.out.println("Begin of try");
            System.out.println(n1/n2);
            System.out.println("End of try");
        }catch(NullPointerException e) {
            System.out.println("Null Pointer Error");
        } catch(ClassCastException e) {
            System.out.println("Cast Error");
        } catch(Exception e) {  //ORDER MATTERS!!
            System.out.println("Undefined Error");
        }
        System.out.println("End of divide method");
}}
```

OUTPUT

```
Start of main method
Start of divide method
Begin of try
Undefined Error
End of divide method
End of main method
```

Getting Info from Exceptions

You may use the following methods with an **exception object** to get valuable information about the exception.

getMessage(): String

- Returns the message that describes this exception object.

toString(): String

- Returns "name of exception class **:** getMessage()".

When to use Exception Handling?

Defensive programming

- It is appropriate when the potential error is predictable and localized
  - e.g. , checking that a queue is not full before adding a new element.

Exception handling

- It is intended for conditions that are …
  - **Unpredictable**: caused by external events out of the control of the program, such as file errors
  - **Widespread**: occurring at many places in the program

Exception handling **separates error-handling code** from normal flow, 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.

Exercise

Suppose that **statement2 causes an exception**:

```
try {
   statement1;
   statement2;
   statement3;
} catch (Exception1 ex1) {
} catch (Exception2 ex2) {
}
statement4;
```

- Will **statement3** run?
- If the exception is not caught, will **statement4** run?
- If the exception is caught in a **catch** block, will **statement4** run?

Exercise

Suppose that **statement2 causes an exception**:

```
try {
   statement1;
   statement2;
   statement3;
} catch (Exception1 ex1) {
} catch (Exception2 ex2) {
} finally {
  statement4;
}
statement5;
```

- Will **statements 4 and 5** be executed if
  - A) the exception is caught in a **catch** block
  - B) the exception is not caught