



Exception Handling in Java

Motivations

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?

This is the subject we introduce in this lecture.

Exception-Handling Overview

(Example 1)

- Review and execute the following programs (provided under samples):
 - *Quotient.java*
 - *QuotientWithIf.java*
 - *QuotientWithException.java*
- Notice that in this program, if you entered 0 for the second number, a runtime error would occur, because you cannot divide an integer by 0

Exception Advantages

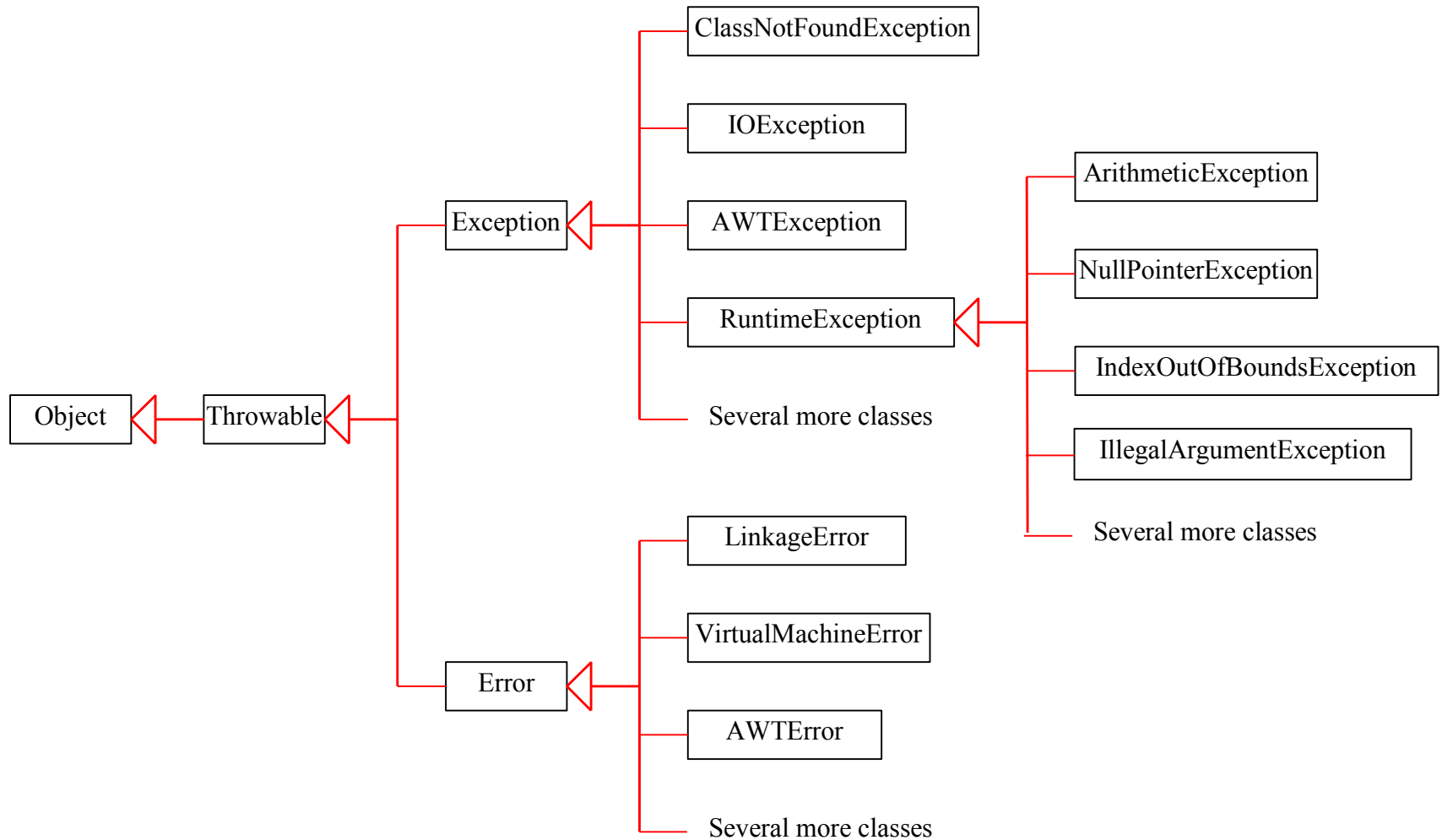
(Example 2 – *QuotientWithMethod*)

Next you see the *advantages* of using exception handling. Review and execute `QuotientWithMethod.java`

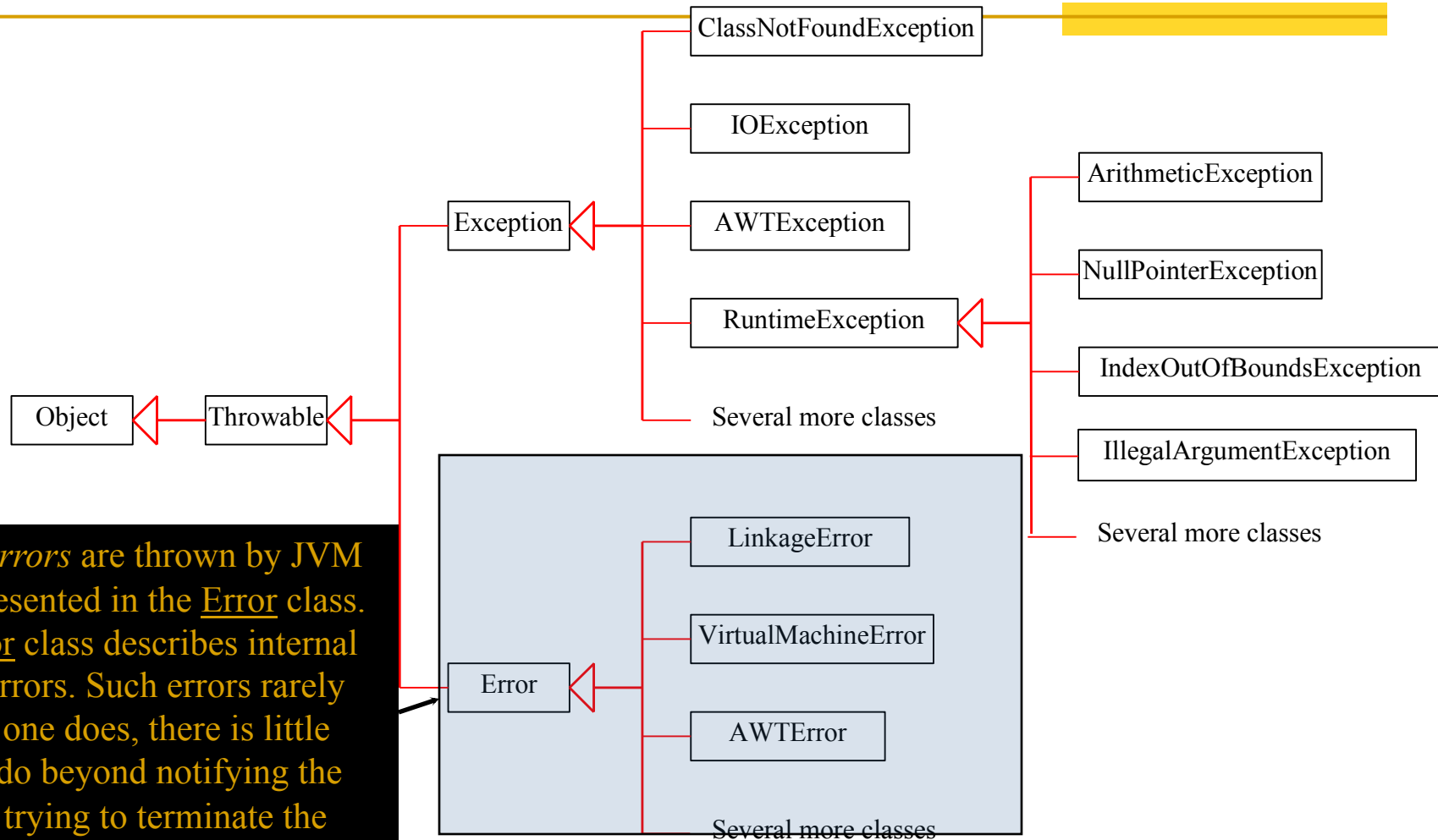
It enables a method to throw an exception to its caller.

Without this capability, a method must handle the exception or terminate the program.

Exception Types



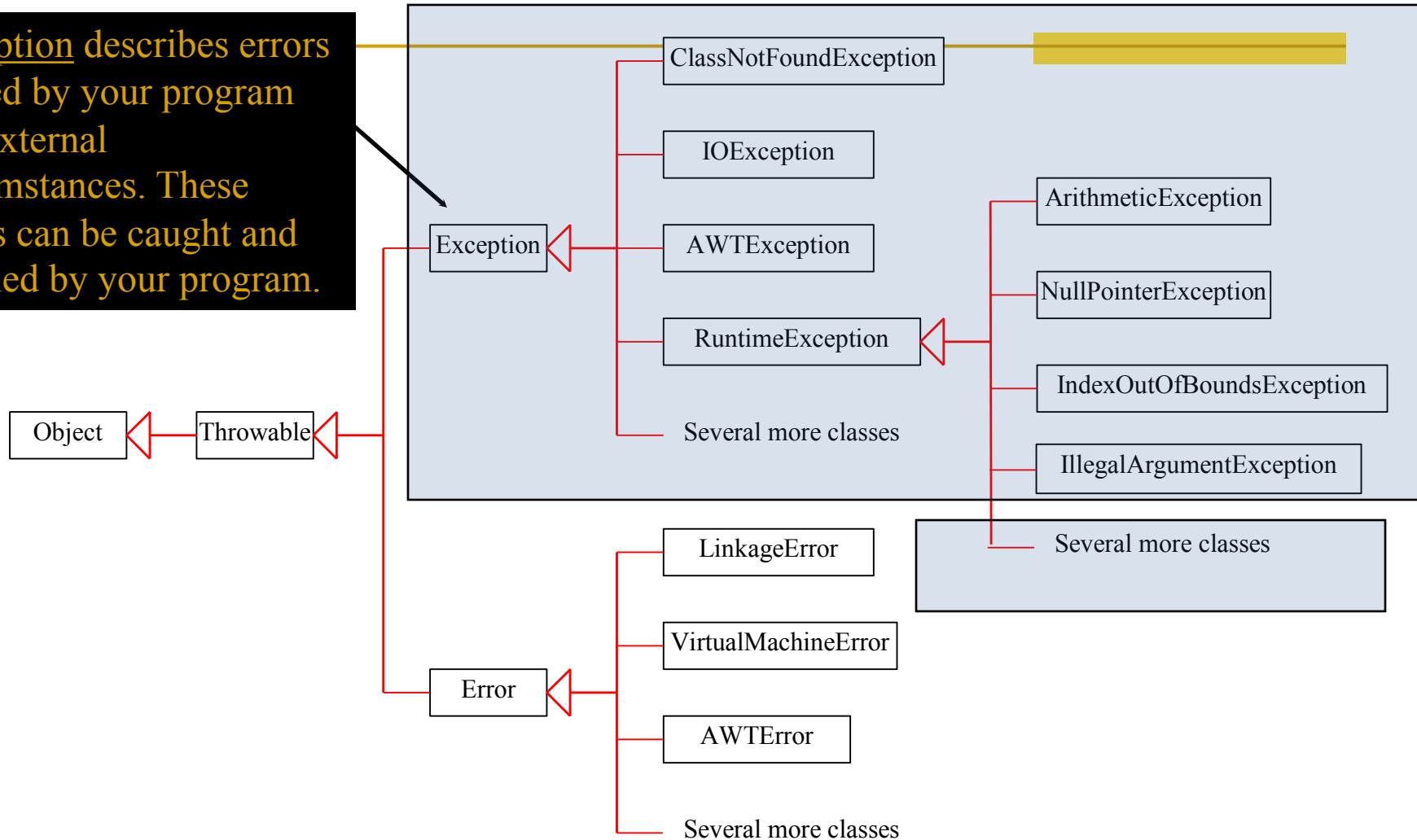
System Errors



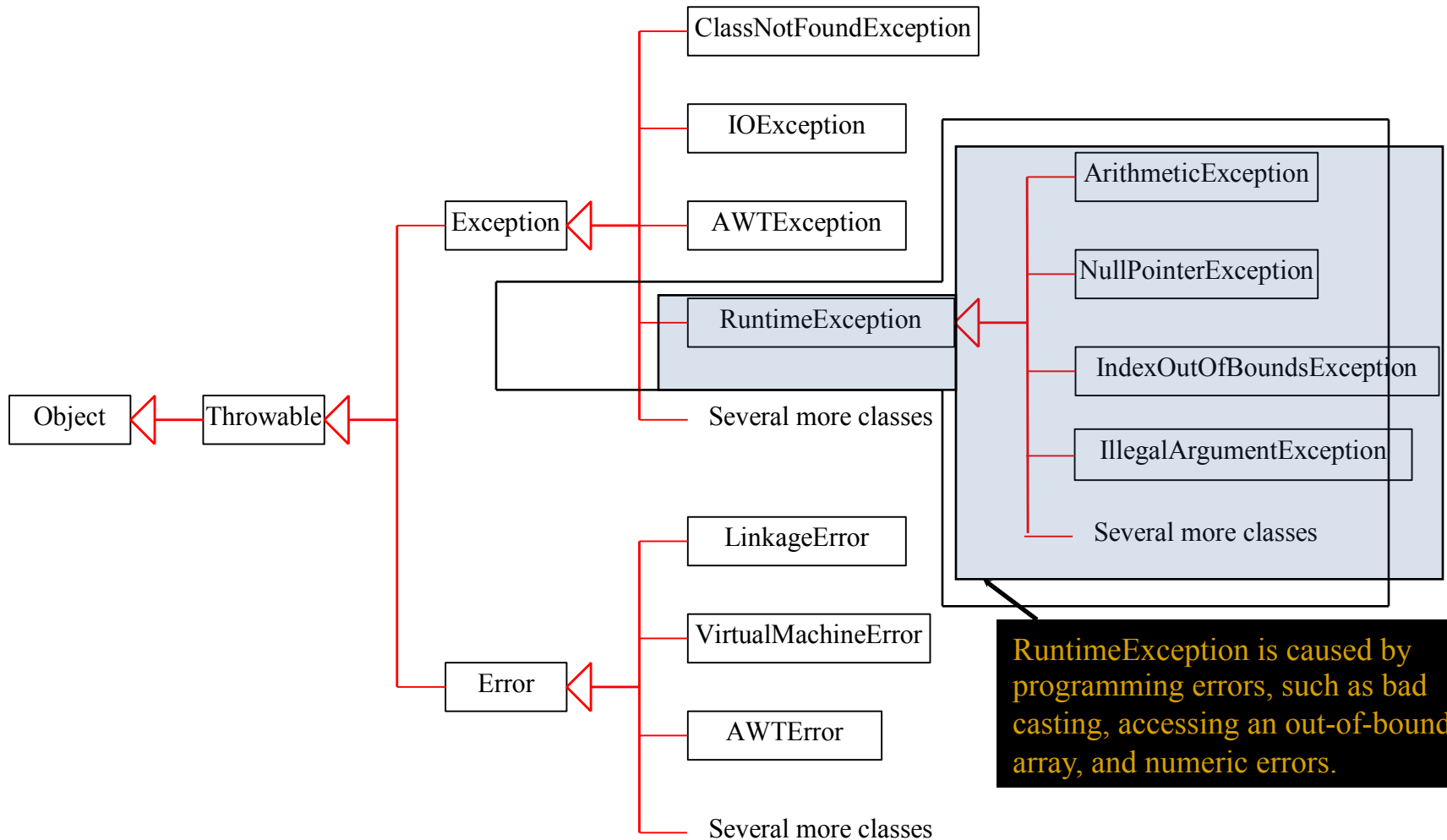
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



Runtime Exceptions



Checked Exceptions vs. Unchecked Exceptions

RuntimeException, Error and their subclasses are known as *unchecked exceptions*.

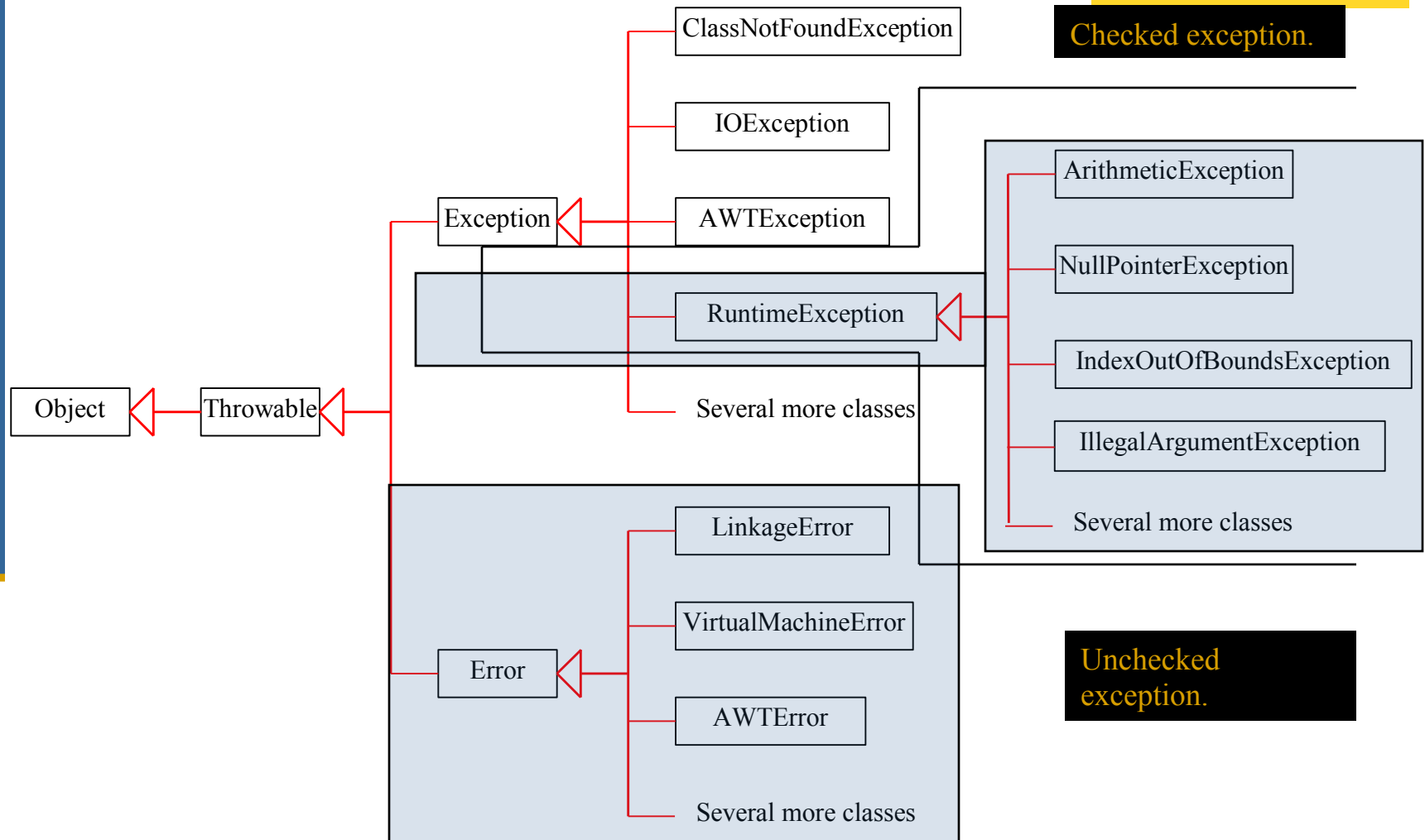
All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Unchecked Exceptions

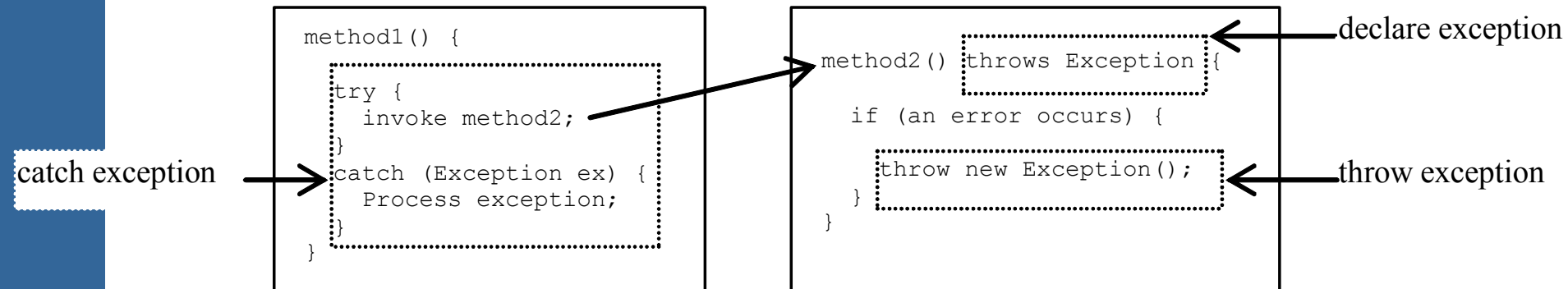
In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example:

- `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it;
- `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array.
 - These are the logic errors that should be corrected in the program.
 - Unchecked exceptions can occur anywhere in the program.
 - To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

Checked or Unchecked Exceptions



Declaring, Throwing, and Catching Exceptions



Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

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

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

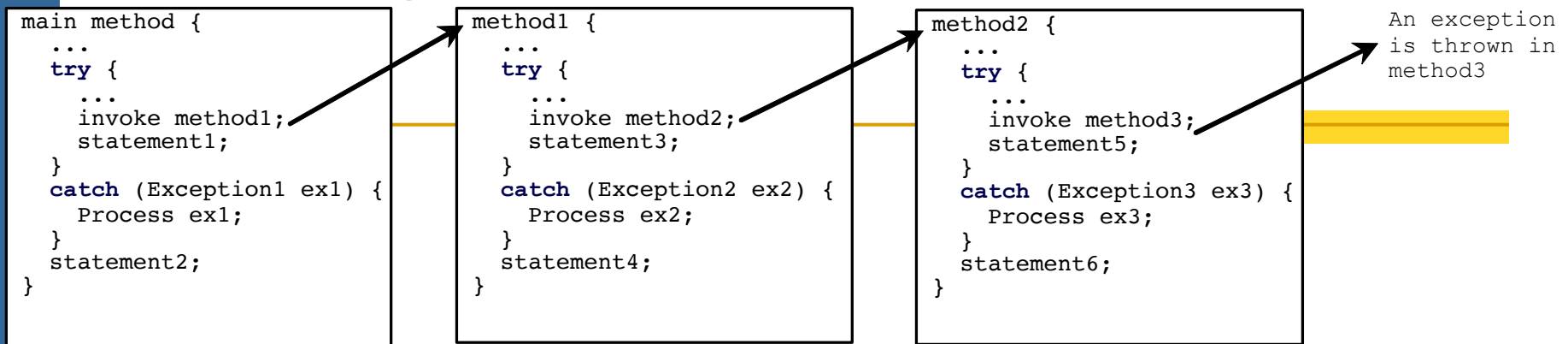
Throwing Exceptions Example

```
/** Set a new radius */  
public void setRadius(double newRadius)  
    throws IllegalArgumentException  
{  
    if (newRadius >= 0)  
        radius = newRadius;  
    else  
        throw new IllegalArgumentException(  
            "Radius cannot be negative");  
}
```

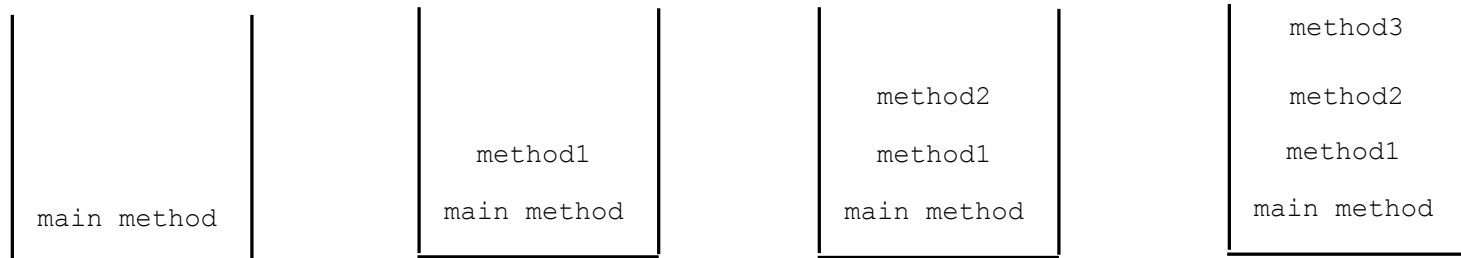
Catching Exceptions

```
try {
    statements;    // Statements that may throw
                  exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
}
catch (Exception2 exVar2) {
    handler for exception2;
}
...
catch (ExceptionN exVar3) {
    handler for exceptionN;
}
```

Catching Exceptions



Call Stack



- ✦ If *method3* cannot handle the exception, *method3* is aborted and the control is returned to *method2*. If exception type is *Exception3* it is caught by the catch block for handling *ex3* in *method2*. *statement5* is skipped and *statement6* is executed.
- ✦ If exception type is *Exception2*, *method2* is aborted and the control is returned to *method1*, and exception is caught by the catch block for handling *ex2* in *method1*. *statement3* is skipped and *statement4* is executed.
- ✦ If exception type is *Exception1*, *method1* is aborted and the control is returned to *main* method, and exception is caught by the catch block for handling *ex1* in the *main* method. *statement1* is skipped and *statement2* is executed.
- ✦ If the exception type is not *Exception1*, *Exception2*, or *Exception3*, the exception is not caught and the program terminates. *statement1* and *statement2* are not executed

Catch or Declare Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {  
    try {  
        p2();  
    }  
    catch (IOException ex) {  
        ...  
    }  
}
```

(a)

```
void p1() throws IOException {  
    p2();  
}
```

(b)

Example 3: Declaring, Throwing, and Catching Exceptions (*TestCircleWithException.java*)

- Objective:

This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class.

The new setRadius method throws an exception if radius is negative.

See sample programs

- CircleWithException.java
- TestCircleWithException.java