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

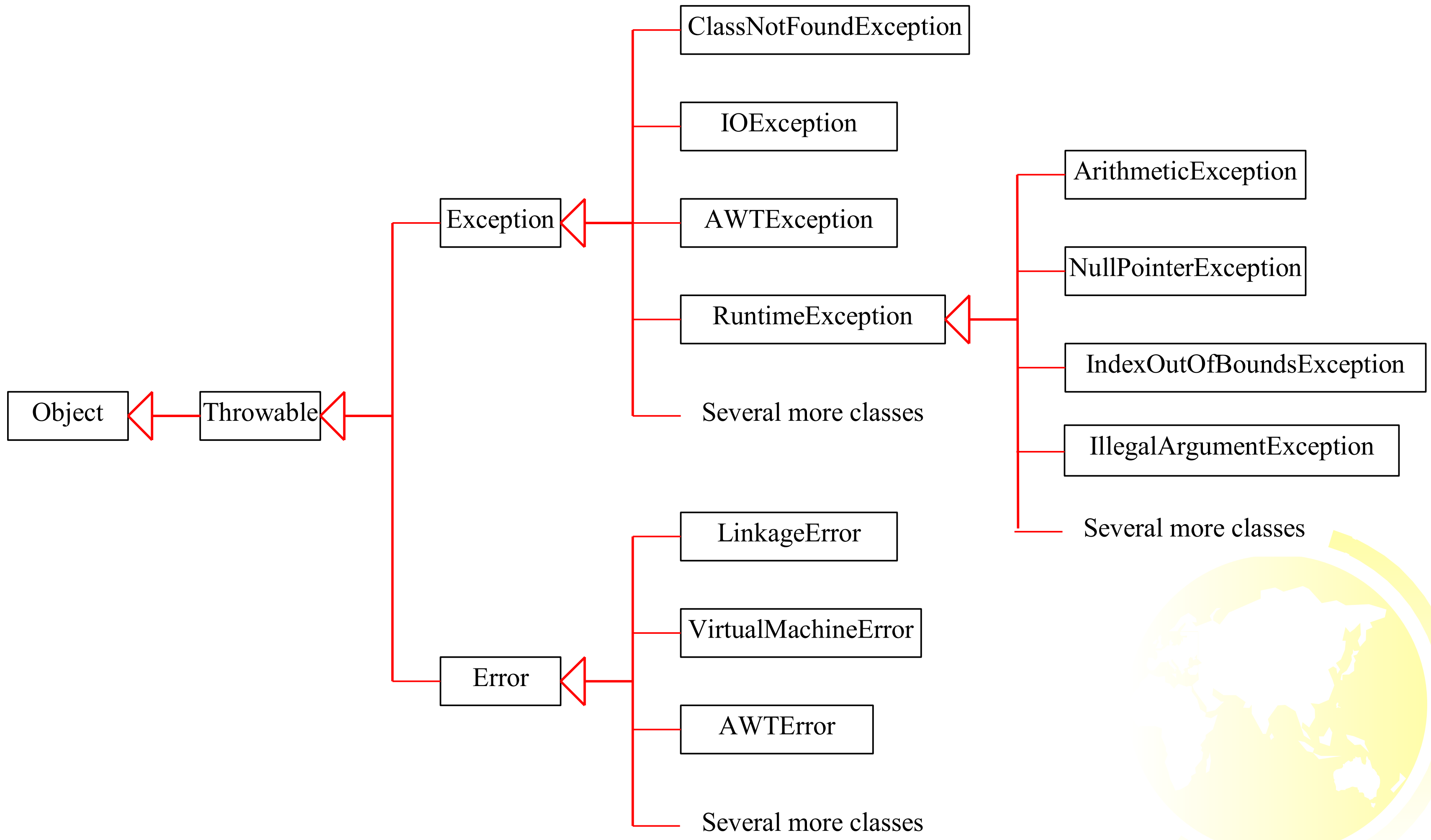
Exception Advantages (Example 2 – *QuotientWithMethod*)

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

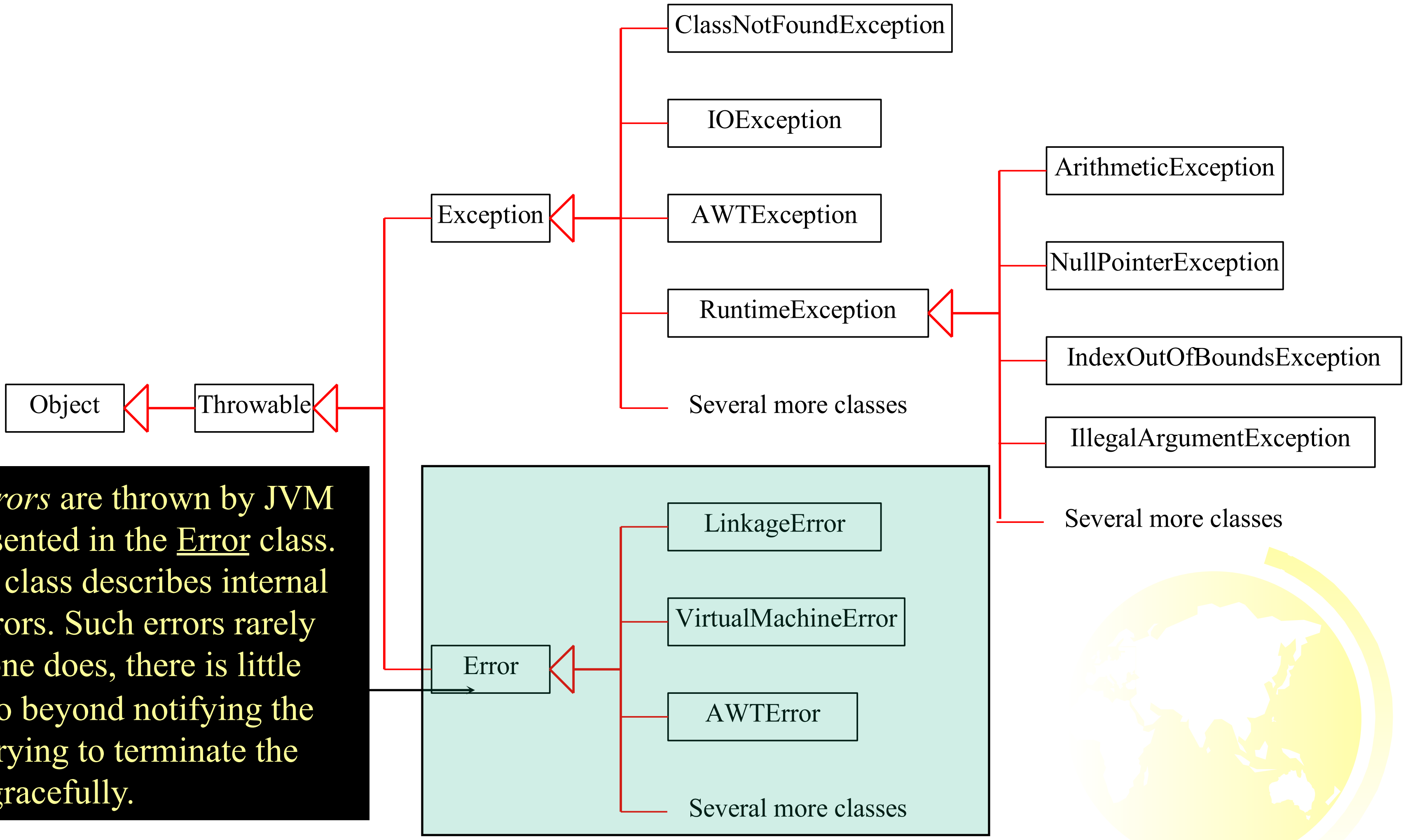
It enables a method to throw an exception to its caller.

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

Exception Types



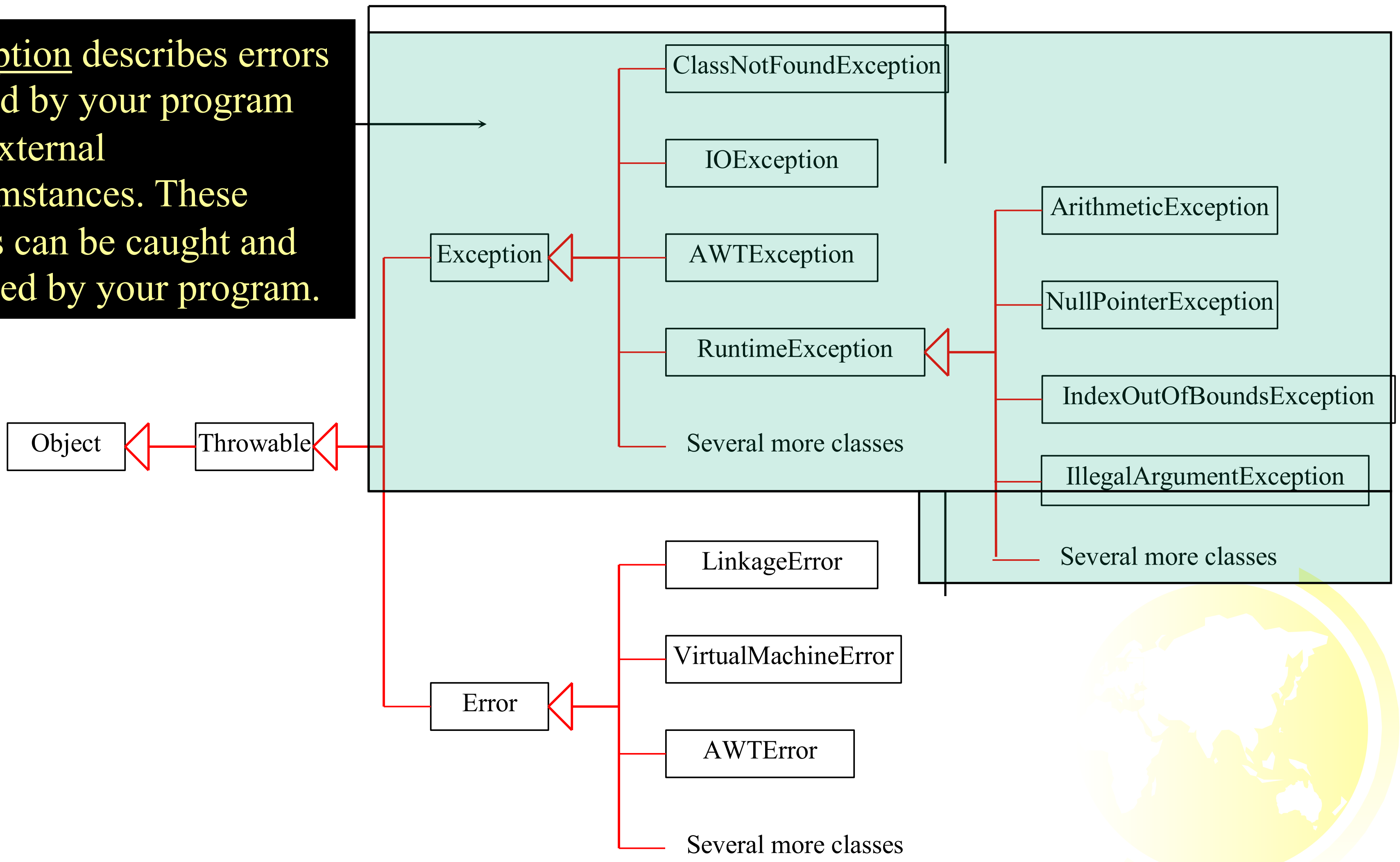
System Errors



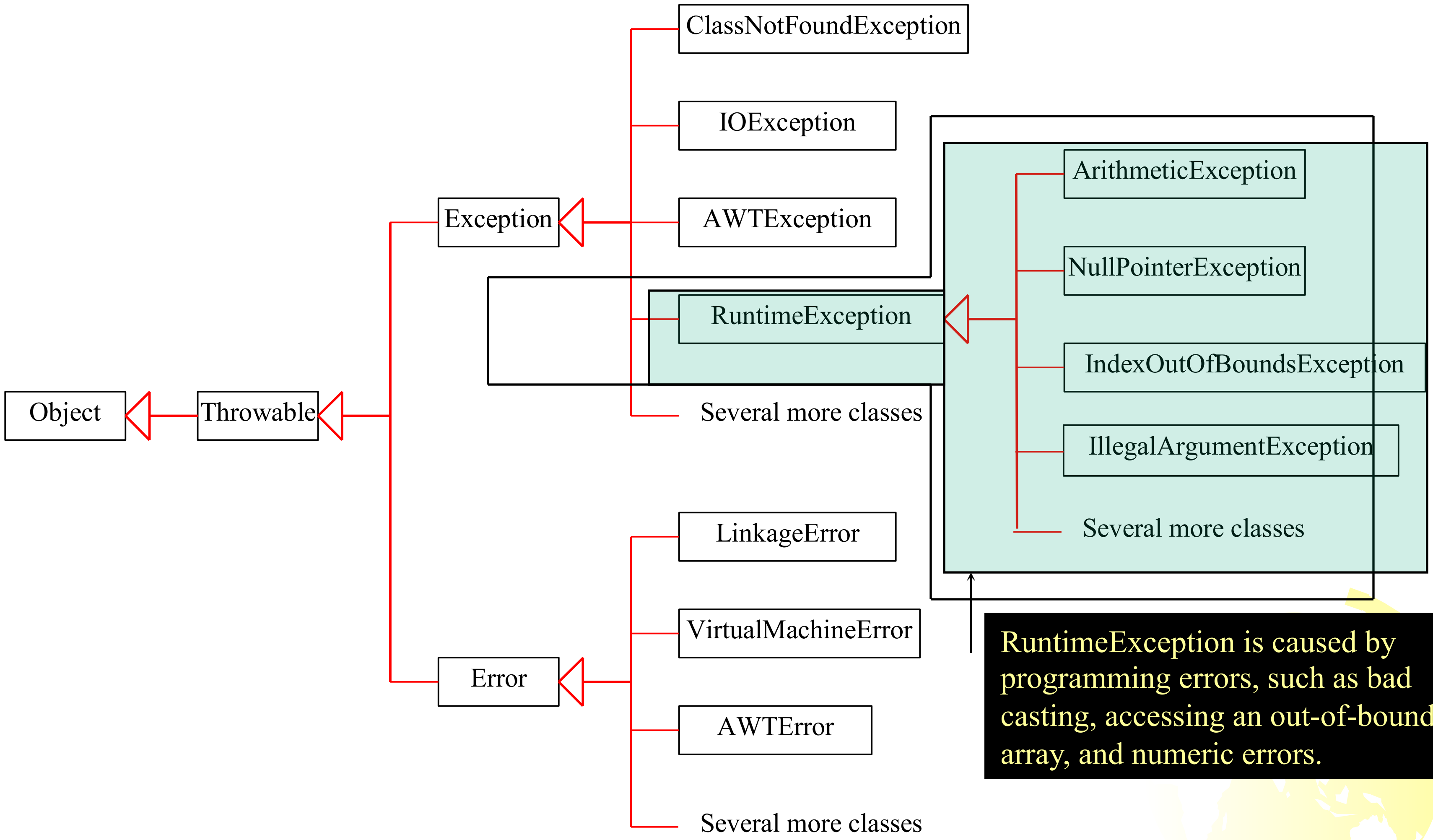
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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



Runtime Exceptions



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Checked Exceptions vs. Unchecked Exceptions

<u>RuntimeException</u>, <u>Error</u> 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

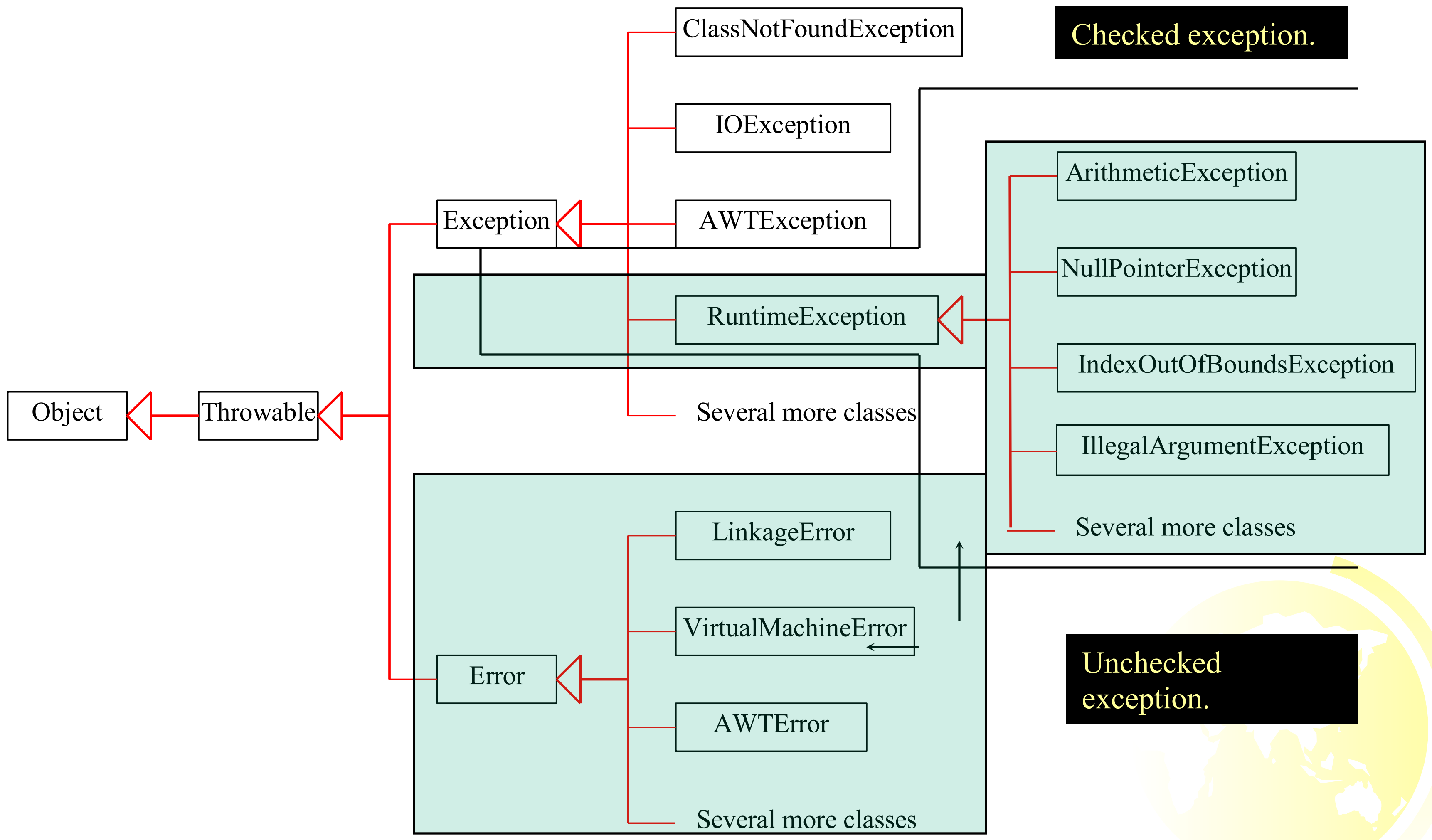
In most cases, unchecked exceptions reflect programming logic errors that are not recoverable.

For example, a <u>NullPointerException</u> is thrown if you access an object through a reference variable before an object is assigned to it;

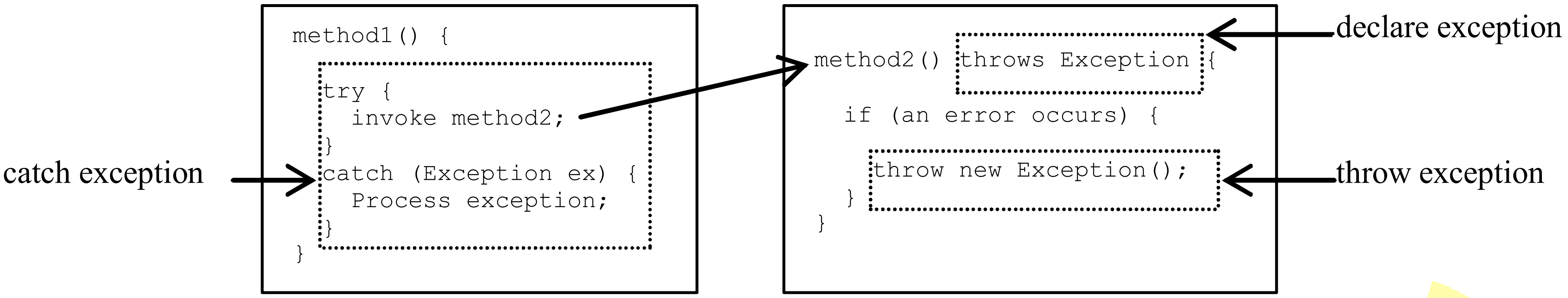
an <u>IndexOutOfBoundsException</u> is thrown if you access an element in an array outside the bounds of the array.

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



```
method1() {

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

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

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
```

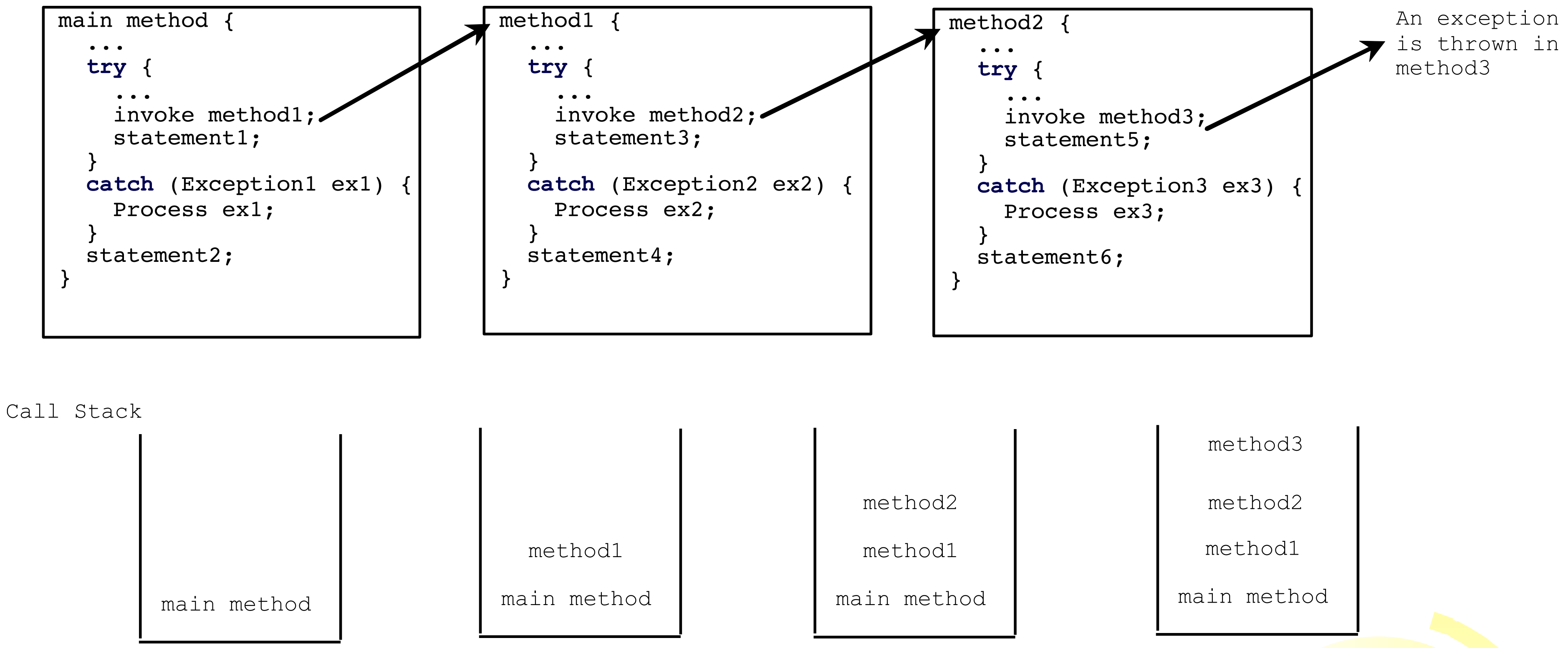
Throwing Exceptions Example

```
/** 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



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

- 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 and TestCircleWithException.java

Rethrowing Exceptions

```
try {
  statements;
}
catch(TheException ex) {
  perform operations before exits;
  throw ex;
}
```

The statement *throw ex* rethrows the exception so that other handlers get a chance to process the exception *ex*.

Sometimes you may need to throw a new exception with additional information along with the original exception. This is called *chained exceptions*.

The `finally` Clause

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

Code in the *finally* block is executed under all circumstances, regardless of whether an exception occurs in the *try* block or is caught.

Trace a Program Execution

Suppose no exceptions in the statements

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

Next statement;
```

Trace a Program Execution

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

Next statement;
```

The final block is always executed

Trace a Program Execution

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

Next statement;
```

Next statement in the method is executed

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

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.

When to Throw Exceptions

- An exception occurs in a method.
- If you want the exception to be processed by its caller, you should create an exception object and throw it.
- If you can handle the exception in the method where it occurs, there is no need to throw it.

When to Use Exceptions

When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

When to Use Exceptions

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Creating Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Create custom exception classes if the predefined classes are not sufficient.
- Declare custom exception classes by extending Exception or a subclass of Exception.

Example 5 - Custom Exception Class (*InvalidRadiusException.java, CircleWithRadiusException.java*)

In previous example, the setRadius method throws an exception if the radius is negative.

Suppose you wish to pass the radius to the handler, you have to create a custom exception class.