COP 3330, Spring 2013

Exceptions Handling

Instructor : Arup Ghosh
02-04-13

School of Electrical Engineering and Computer Science
University of Central Florida

Objectives

- To know what is exception and what is exception handling .
- To distinguish exception types: Error (fatal) vs. Exception (non-fatal), and checked vs. uncheck exceptions.
- To declare exceptions in the method header.
- To throw exceptions out of a method.
- To write a try-catch block to handle exceptions.
- To explain how an exception is propagated.
- To rethrow exceptions in a try-catch block.
- To use the finally clause in a try-catch block.
- To know when to use exceptions.
- To declare custom exception classes.
- To apply assertions to help ensure program correctness.

Syntax Errors, Runtime Errors, and Logic Errors

There are three categories of errors: syntax errors, runtime errors, and logic errors.

Syntax errors arise because the rules of the language have not been followed. They are detected by the compiler.

Syntax Errors, Runtime Errors, and Logic Errors

Runtime errors occur while the program is running if the environment detects an operation that is impossible to carry out.

Logic errors occur when a program doesn't perform the way it was intended to.

Runtime Errors

```
1   import java.util.Scanner;
2
3   public class ExceptionDemo {
4     public static void main(String[] args) {
5       Scanner scanner = new Scanner(System.in);
6       System.out.print("Enter an integer: ");
7       int number = scanner.nextInt();
8
9       // Display the result
10      System.out.println(
11        "The number entered is " + number);
12    }
13  }
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

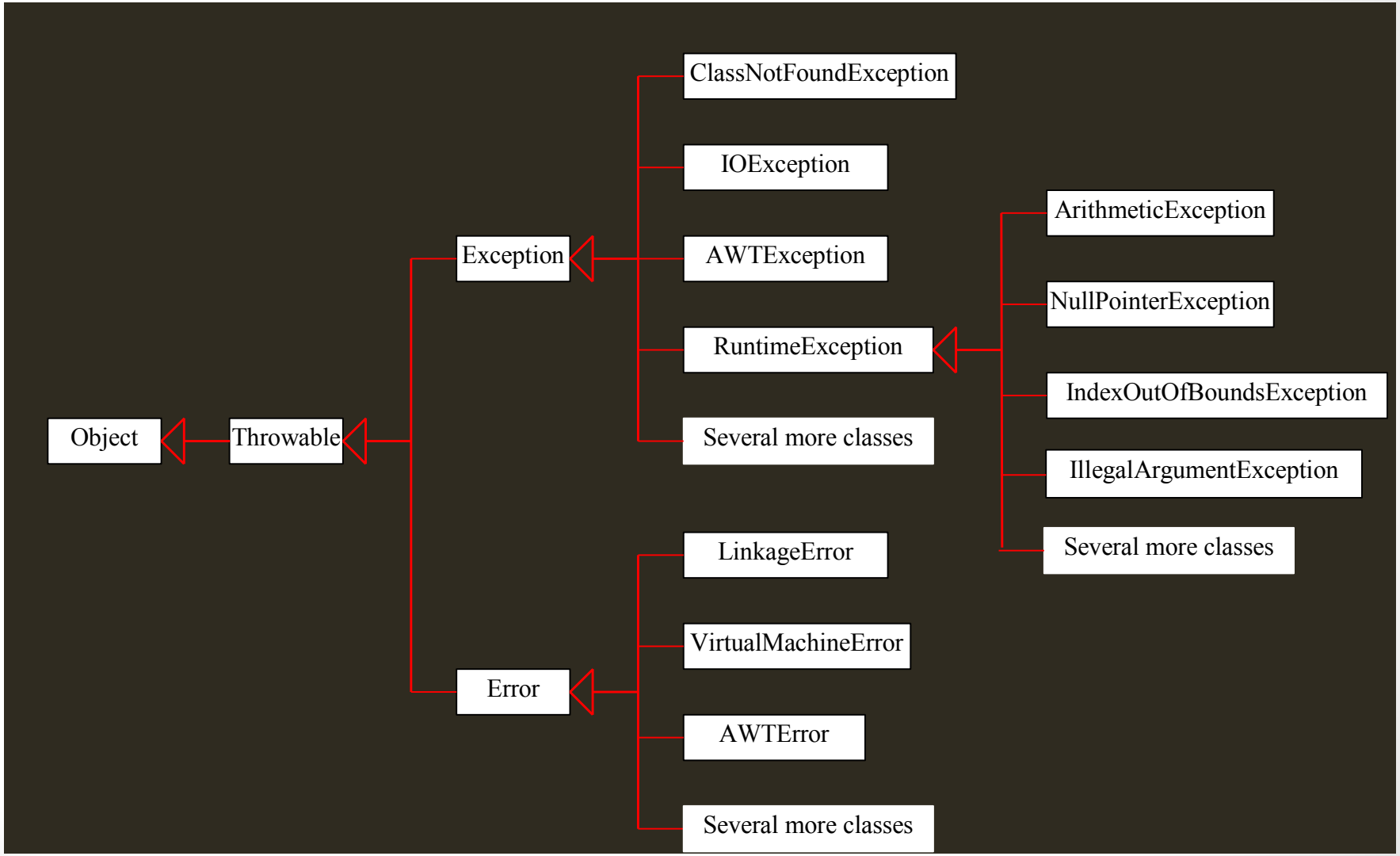
Terminated.

Catch Runtime Errors

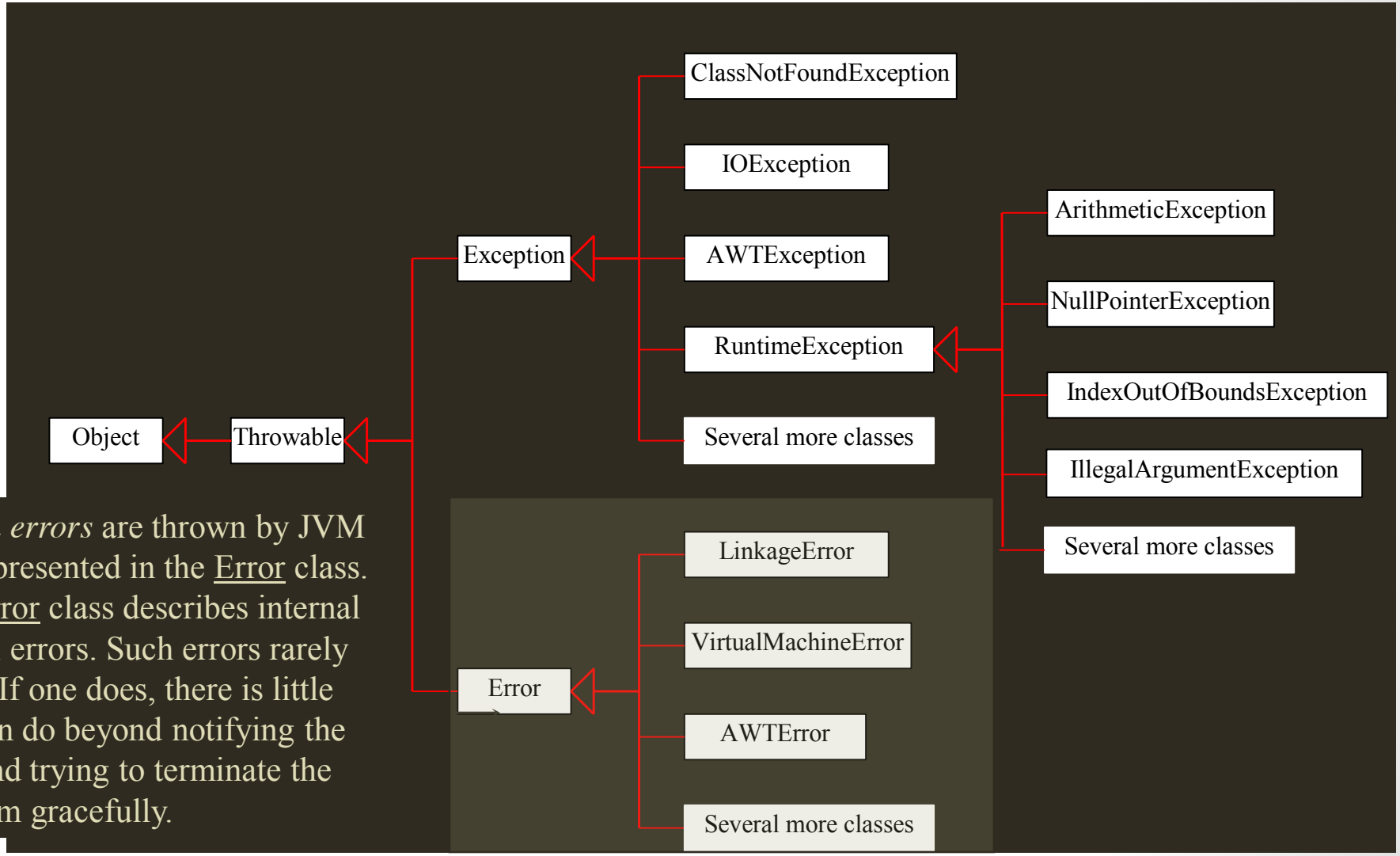
```
1   import java.util.*;
2
3   public class HandleExceptionDemo {
4     public static void main(String[] args) {
5       Scanner scanner = new Scanner(System.in);
6       boolean continueInput = true;
7
8       do {
9         try {
10          System.out.print("Enter an integer: ");
11          int number = scanner.nextInt();
12
13          // Display the result
14          System.out.println(
15            "The number entered is " + number);
16
17          continueInput = false;
18        }
19        catch (InputMismatchException ex) {
20          System.out.println("Try again. (" +
21            "Incorrect input: an integer is required)");
22          scanner.nextLine(); // discard input
23        }
24      } while (continueInput);
25    }
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

Exception Classes



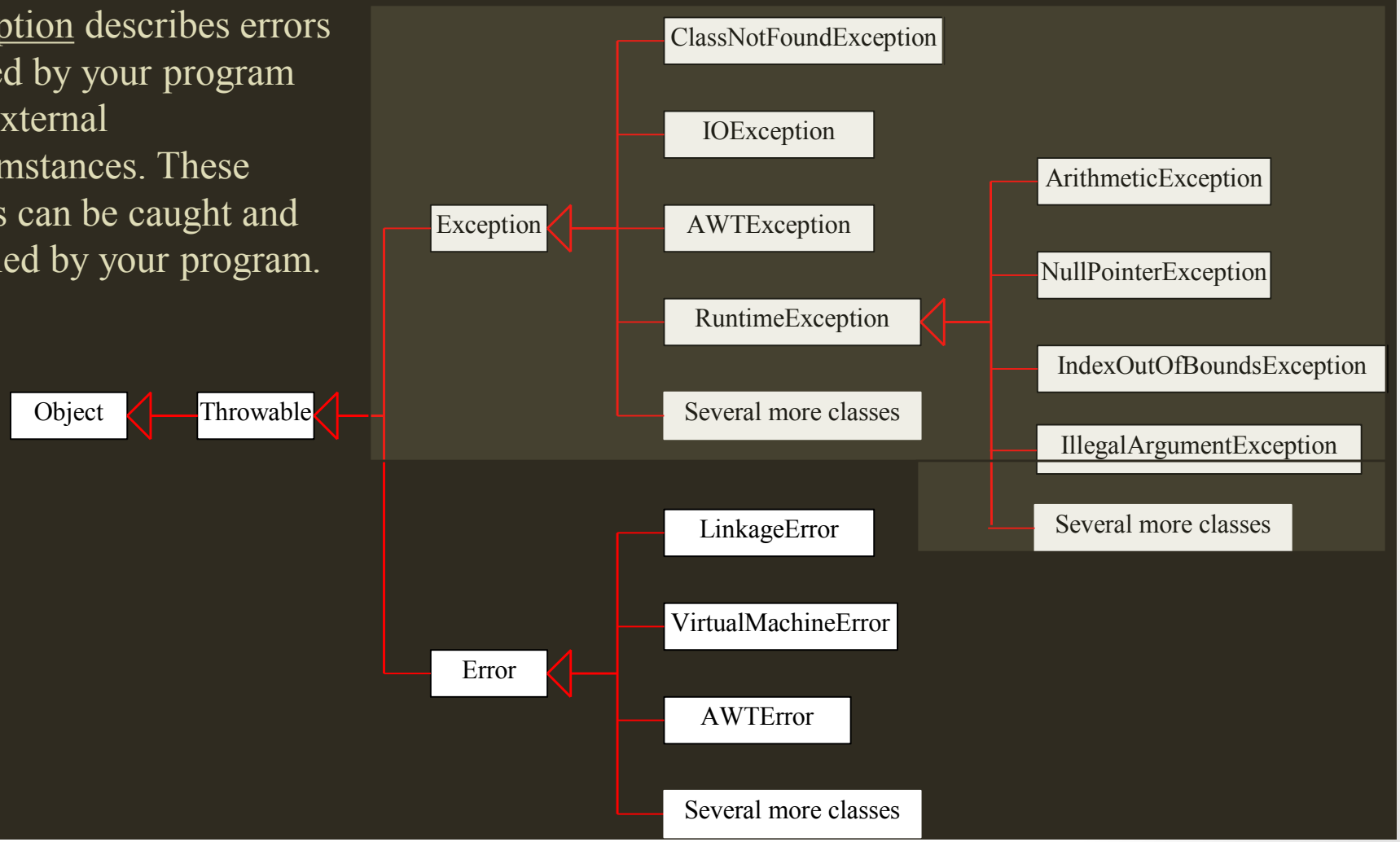
System Errors



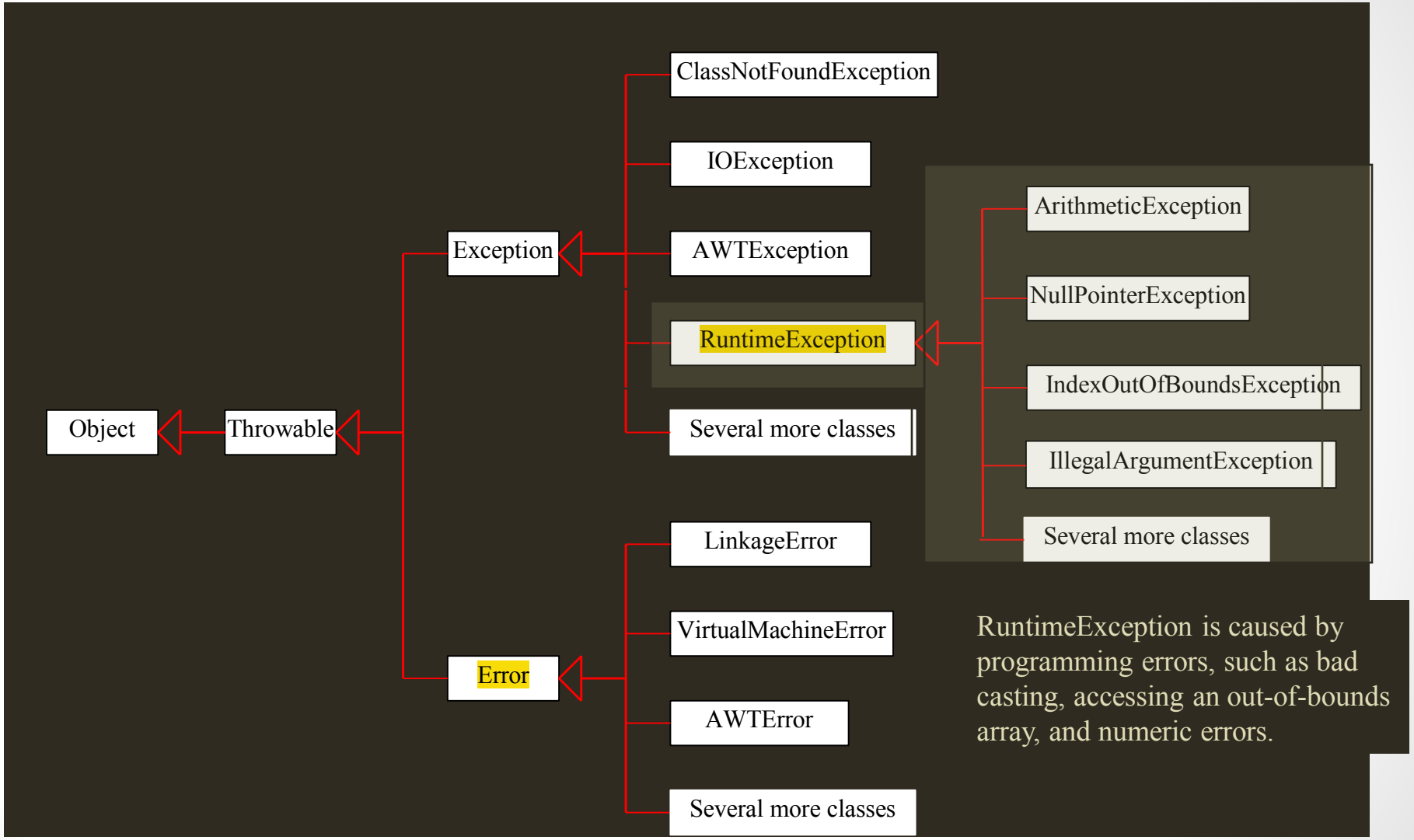
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

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, 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.

Java Exception Hierarchy

- Checked exceptions vs. unchecked exceptions.
  - Compiler enforces a catch-or-declare requirement for checked exceptions.
- An exception's type determines whether it is checked or unchecked.
- Direct or indirect subclasses of class `RuntimeException` (package `java.lang`) are *unchecked* exceptions.
  - Typically caused by defects in your program's code (e.g., `ArrayIndexOutOfBoundsException`s).
- Subclasses of `Exception` but not `RuntimeException` are *checked* exceptions.
  - Caused by conditions that are not in the control of the program—e.g., in file processing, the program can't open a file because the file does not exist.

Java Exception Hierarchy

- Classes that inherit from class `Error` are considered to be *unchecked*.
- The compiler *checks* each method call and method declaration to determine whether the method throws checked exceptions.
  - If so, the compiler verifies that the checked exception is caught or is declared in a `throws` clause.
- `throws` clause specifies the exceptions a method throws.
  - Such exceptions are typically not caught in the method's body.

Java Exception Hierarchy

- To satisfy the *catch* part of the *catch-or-declare requirement*, the code that generates the exception must be wrapped in a `try` block and must provide a `catch` handler for the checked-exception type (or one of its superclasses).
- To satisfy the *declare* part of the *catch-or-declare requirement*, the method must provide a `throws` clause containing the checked-exception type after its parameter list and before its method body.
- If the catch-or-declare requirement is not satisfied, the compiler will issue an error message indicating that the exception must be caught or declared.

Java Exception Hierarchy

- The compiler does not check the code to determine whether an unchecked exception is caught or declared.
  - These typically can be prevented by proper coding.
  - For example, an `ArithmeticException` can be avoided if a method ensures that the denominator is not zero before attempting to perform the division.
- Unchecked exceptions are not required to be listed in a method's `throws` clause.
  - Even if they are, it's not required that such exceptions be caught by an application.

`finally` Block

- `finally` block will execute whether or not an exception is thrown in the corresponding `try` block.
- `finally` block will execute if a `try` block exits by using a `return`, `break` or `continue` statement or simply by reaching its closing right brace.
- `finally` block will *not* execute if the application terminates immediately by calling method `System.exit`.

`finally` Block

- Both `System.out` and `System.err` are streams—a sequence of bytes.
  - `System.out` (the standard output stream) displays output
  - `System.err` (the standard error stream) displays errors
- Output from these streams can be redirected (e.g., to a file).
- Using two different streams enables you to easily separate error messages from other output.
  - Data output from `System.err` could be sent to a log file
  - Data output from `System.out` can be displayed on the screen

`finally` Block

- `throw` statement—indicates that an exception has occurred.
  - Used to throw exceptions.
  - Indicates to client code that an error has occurred.
  - Specifies an object to be thrown.
  - The operand of a `throw` can be of any class derived from class `Throwable`.

The `finally` Clause

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

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