

Exception Handling

Introduction to Programming and
Computational Problem Solving - 2

CSE 8B

Lecture 11

Announcements

- Assignment 5 is due May 6, 11:59 PM
- Assignment 6 will be released May 9
 - Due May 16, 11:59 PM
- Reading
 - Liang
 - Chapter 12

Exceptions

- Exceptions are runtime errors caused by your program and external circumstances
 - These errors can be caught and handled by your program

Example: integer divide by zero

```
import java.util.Scanner;

public class Quotient {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        System.out.println(number1 + " / " + number2 + " is " +
            (number1 / number2));
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

Example: integer divide by zero

- Exception in thread "main" java.lang.ArithmeticException: / by zero
- First approach
 - Mitigate exception with if statement
 - Create a method, so we can reuse it

Example: integer divide by zero

```
import java.util.Scanner;

public class QuotientWithMethod {
    public static int quotient(int number1, int number2) {
        if (number2 == 0) {
            System.out.println("Divisor cannot be zero");
            System.exit(1);
        }
        return number1 / number2;
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        int result = quotient(number1, number2);
        System.out.println(number1 + " / " + number2 + " is "
            + result);
    }
}
```

Problem:
a method should
never terminate
a program

Example: integer divide by zero

- Exception in thread "main" java.lang.ArithmeticException: / by zero
- First approach
 - Mitigate exception with if statement
 - Create a method, so we can reuse it
 - Problem: a method should never terminate a program
- Second approach
 - Have the method notify the caller

Example: integer divide by zero

```
import java.util.Scanner;

public class QuotientWithException {
    public static int quotient(int number1, int number2) {
        if (number2 == 0)
            throw new ArithmeticException("Divisor cannot be zero");

        return number1 / number2;
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        try {
            int result = quotient(number1, number2);
            System.out.println(number1 + " / " + number2 + " is "
                + result);
        }
        catch (ArithmeticException ex) {
            System.out.println("Exception: an integer " +
                "cannot be divided by zero ");
        }

        System.out.println("Execution continues ...");
    }
}
```

Throw an
ArithmeticException

Try something that may
throw an exception

Catch an
ArithmeticException

Handle the caught
exception in the catch block

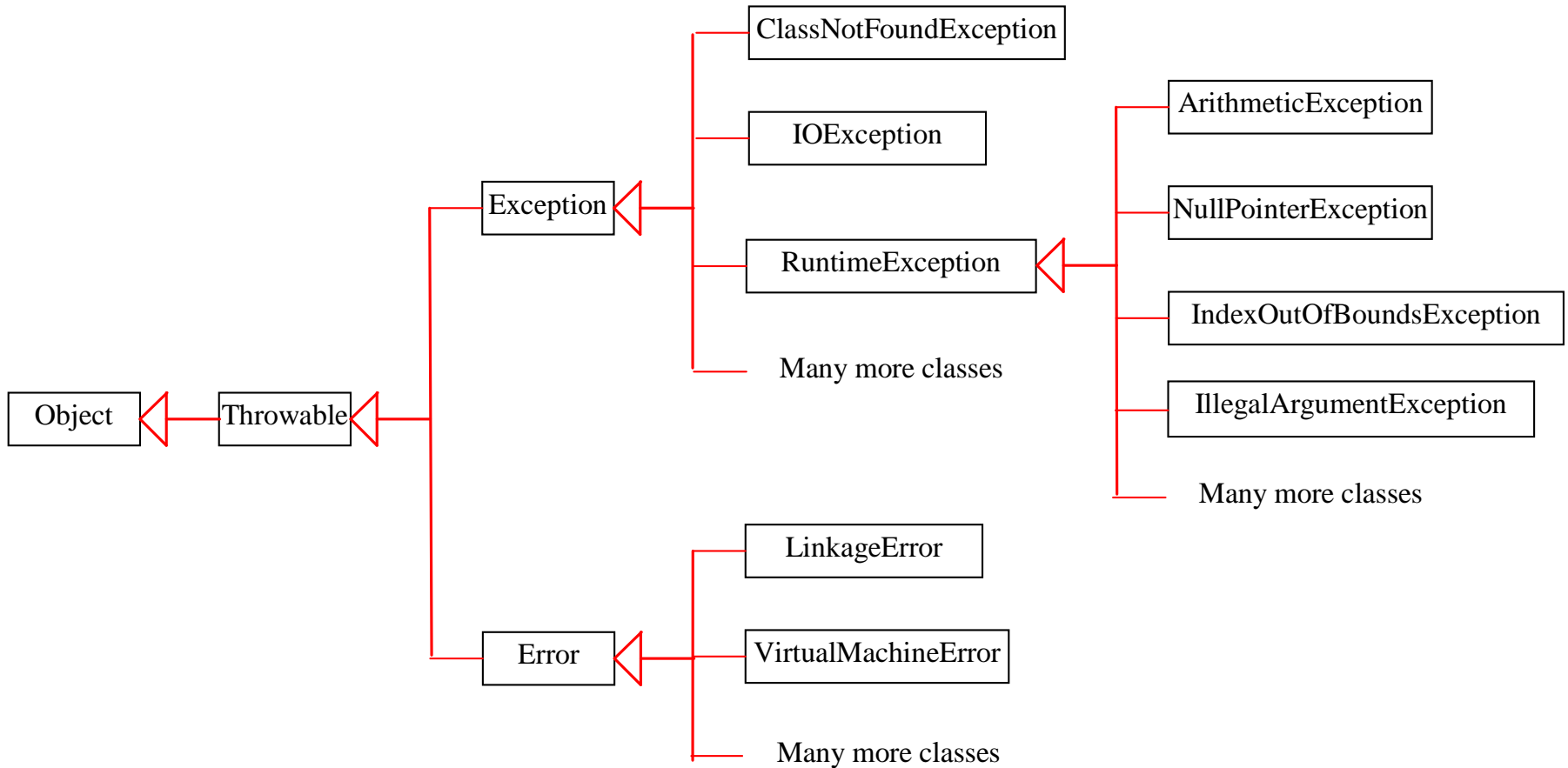
Exception handling

- Exception handling enables a method to throw an exception to its caller
- Without this capability, a method must handle the exception or terminate the program
- Separates
 - The detection of an error
 - The handling of an error

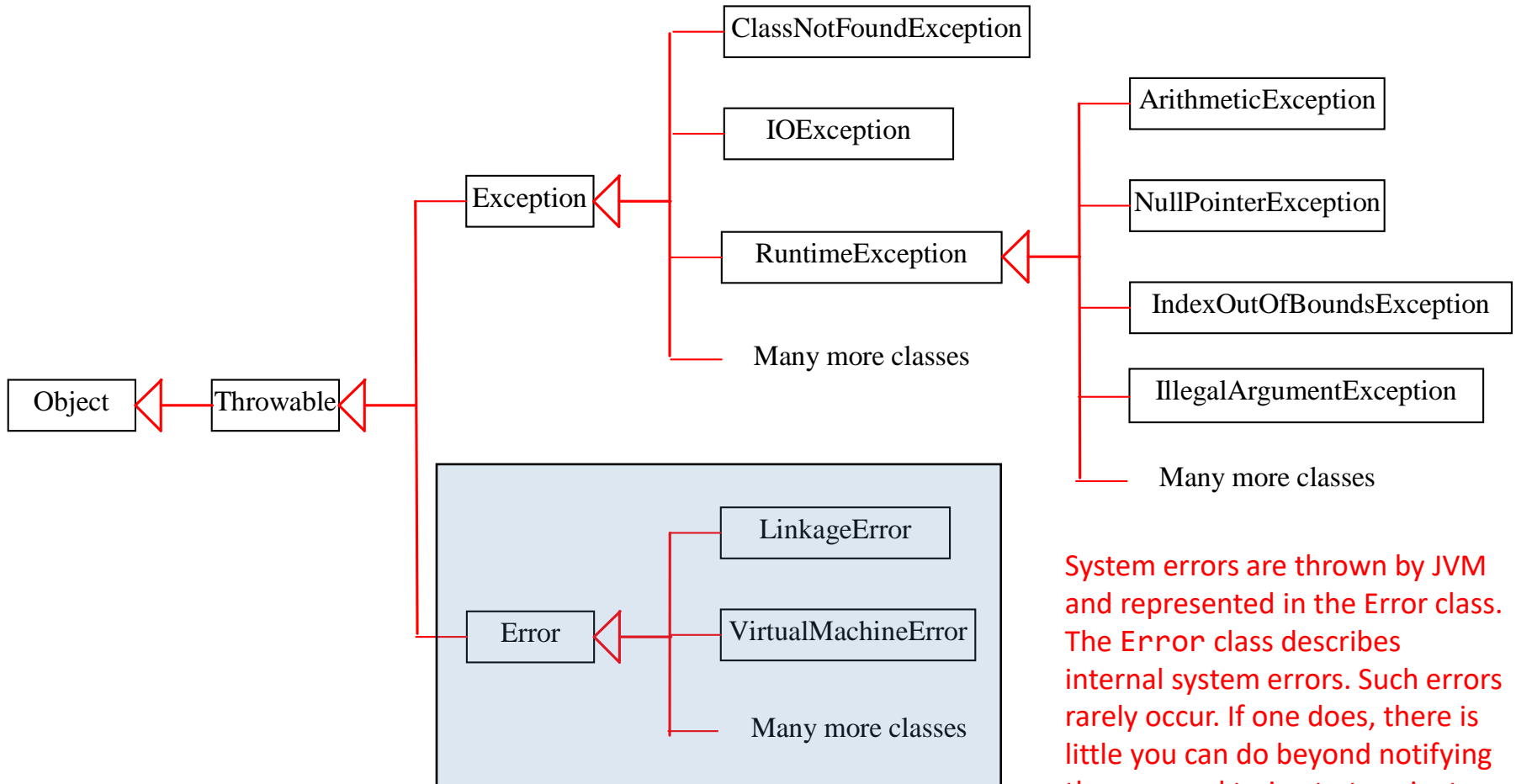
Exception types

- Exceptions are objects
 - Remember, objects are instances of classes
- The root class for exception is `java.lang.Throwable`
 - Three major types
 - System errors
 - Exceptions
 - Runtime Exceptions

Exception types



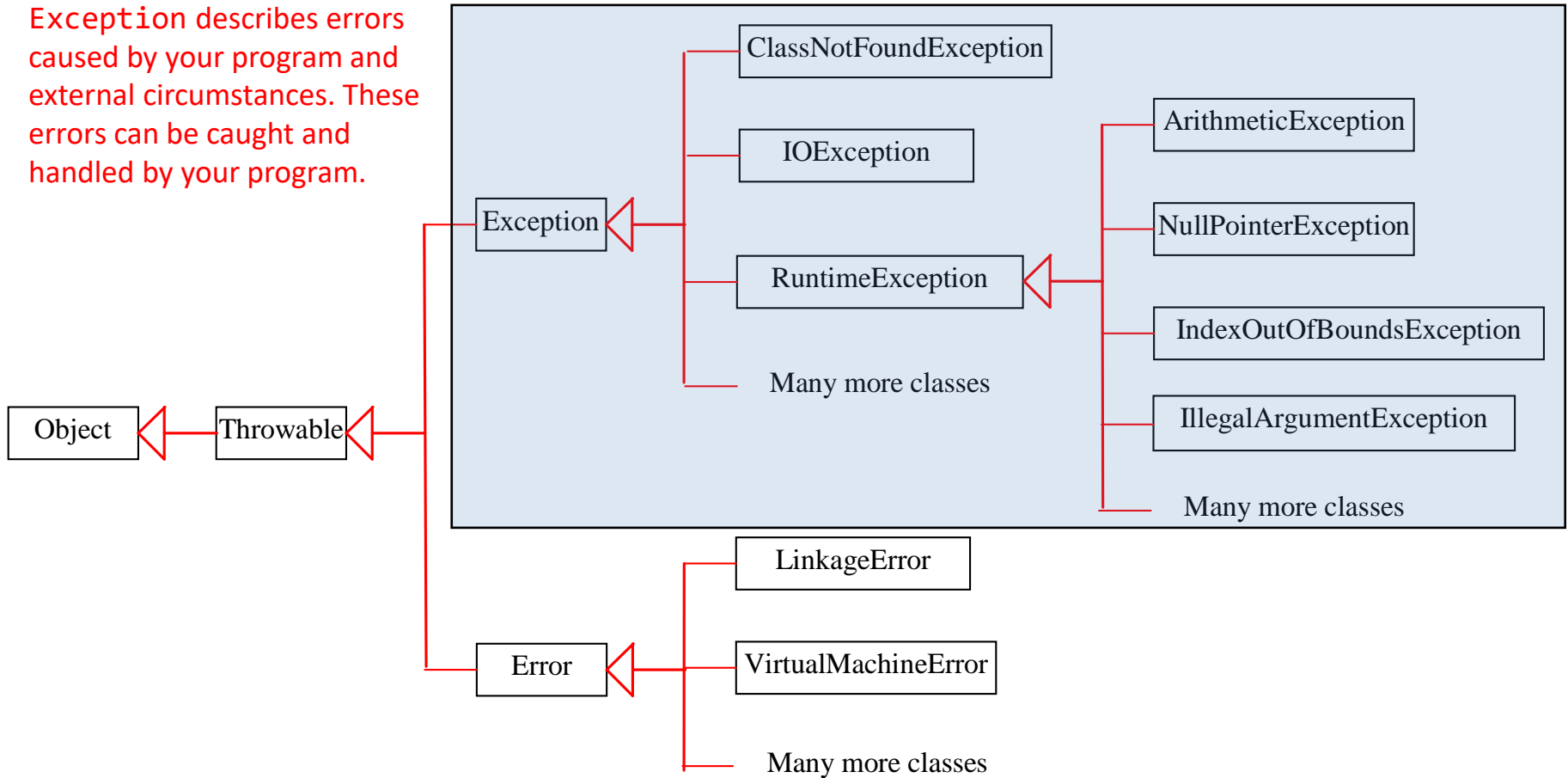
Error



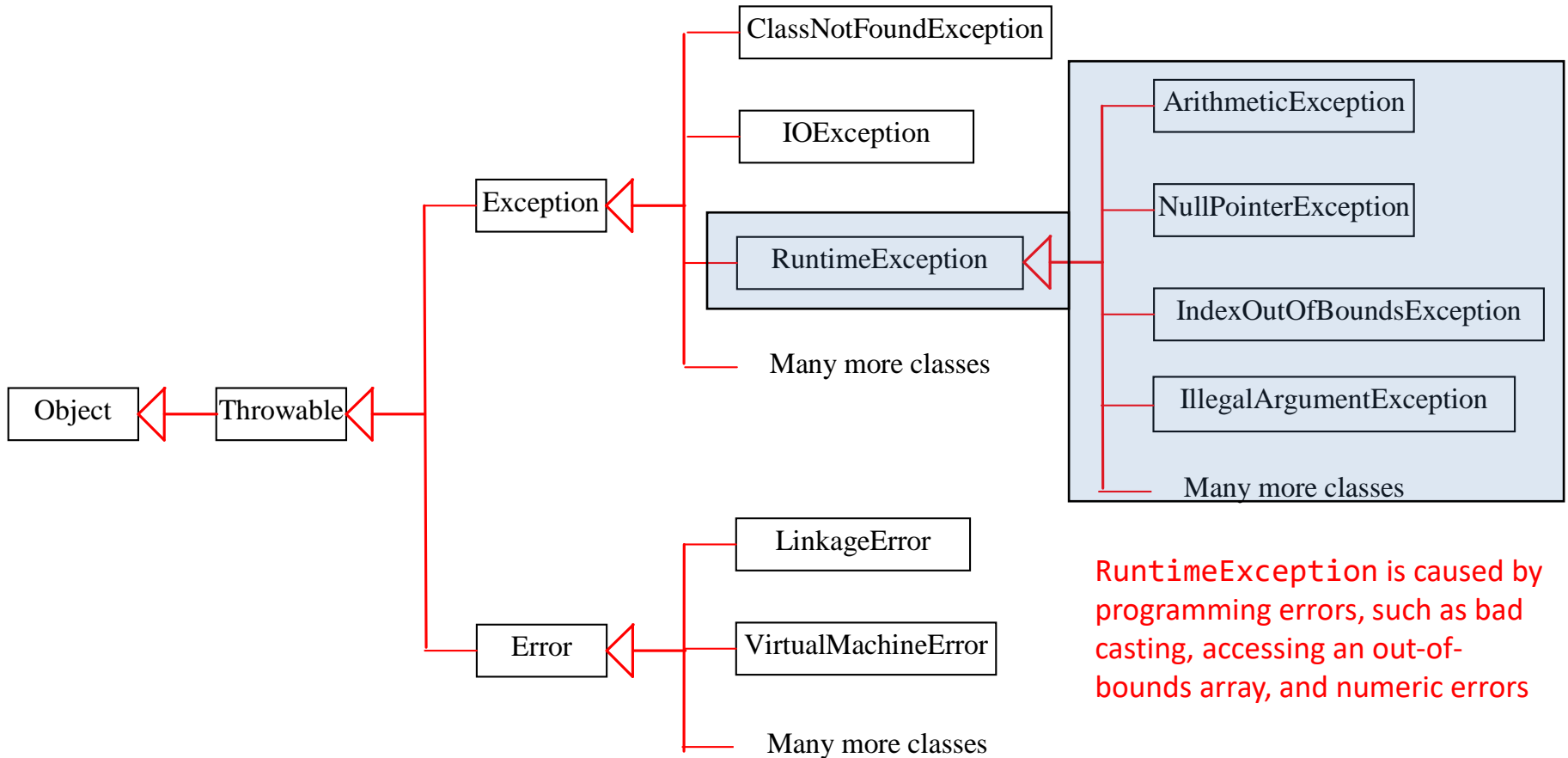
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exception

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



RuntimeException



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

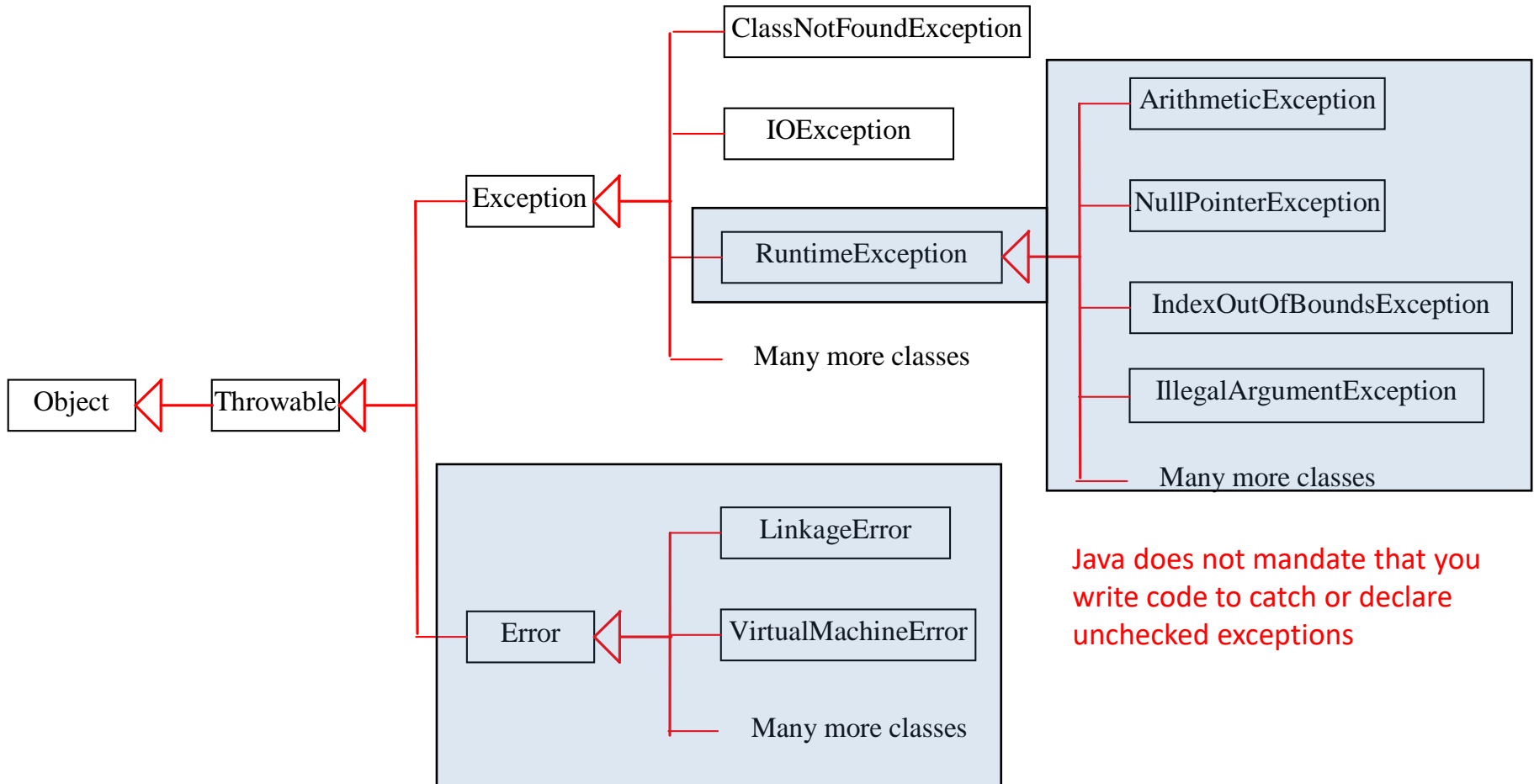
Exception types

- Exceptions are objects
 - Remember, objects are instances of classes
- The root class for exception is `java.lang.Throwable`
 - All Java exception classes inherit directly or indirectly from `Throwable`
- You can create your own exception classes by extending `Exception` or a subclass of `Exception`

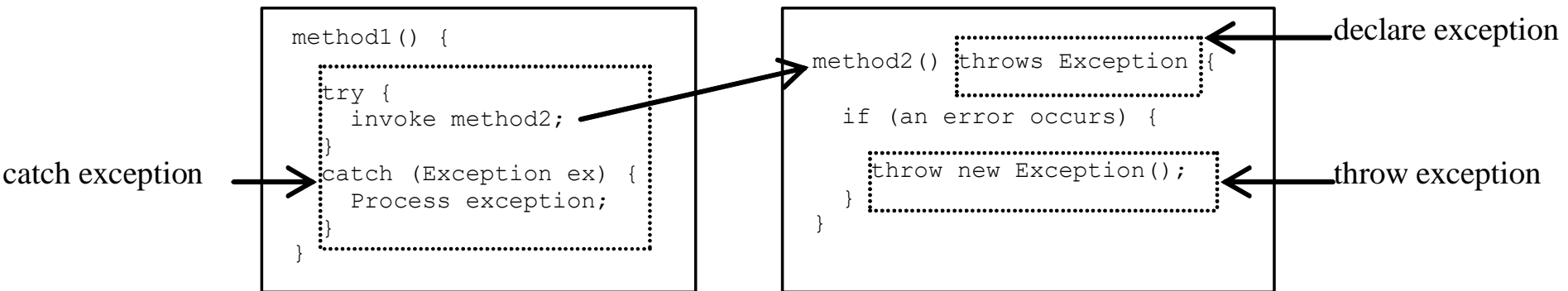
Unchecked exceptions vs. checked exceptions

- `RuntimeException`, `Error`, and their subclasses are known as *unchecked exceptions*
 - Usually programming logic errors that are unrecoverable
 - These should be corrected in the program
- All other exceptions are known as *checked exceptions*
 - **The compiler forces the programmer to check and deal with these exceptions**

Unchecked exceptions



Declaring, throwing, and catching exceptions



Declaring exceptions

- Every method must state the types of *checked exceptions* it might throw
 - This is called *declaring exceptions*

- Examples

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

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

Throwing exceptions

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it
 - This is called *throwing an exception*

- For 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

- When an exception is thrown, it can be caught and handled in a `try-catch` block
 - If no exceptions are thrown in the `try` block, then the catch blocks are skipped
- If an exception is thrown in the `try` block, Java **skips the remaining statements in the `try` block** and starts the process of finding the code to handle the exception
 - This is called *catching an exception*

Catching exceptions

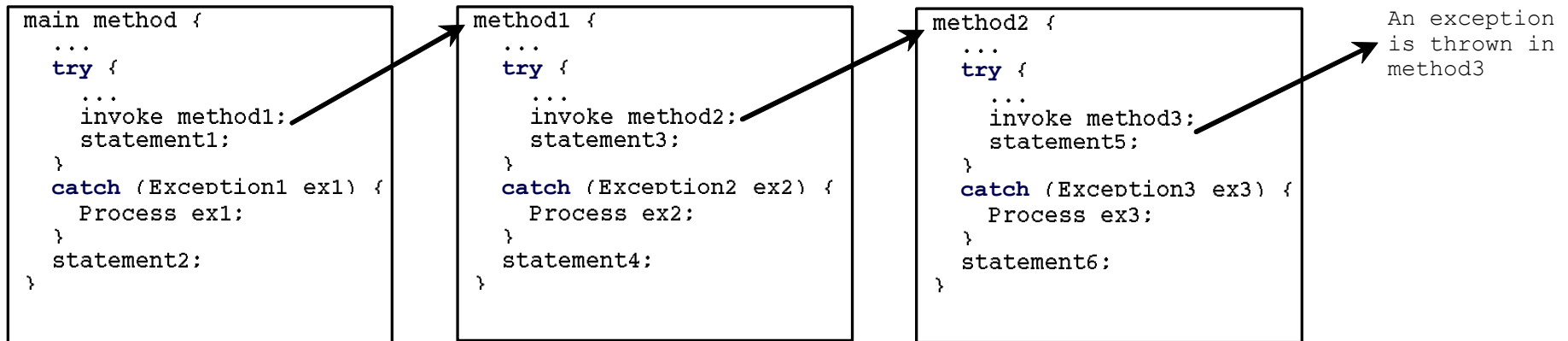
```
try {  
    // Statements that may throw exceptions  
}  
catch (Exception1 exVar1) {  
    // Handler for Exception1  
}  
catch (Exception2 | Exception3 | ... | ExceptionK exVar)  
{  
    // Same code for handling these exceptions  
}  
...  
catch (ExceptionN exVarN) {  
    // Handler for ExceptionN  
}
```

The order exceptions are specified is important. A compile error occurs if a catch block for a superclass type appears before a catch block for a subclass type.

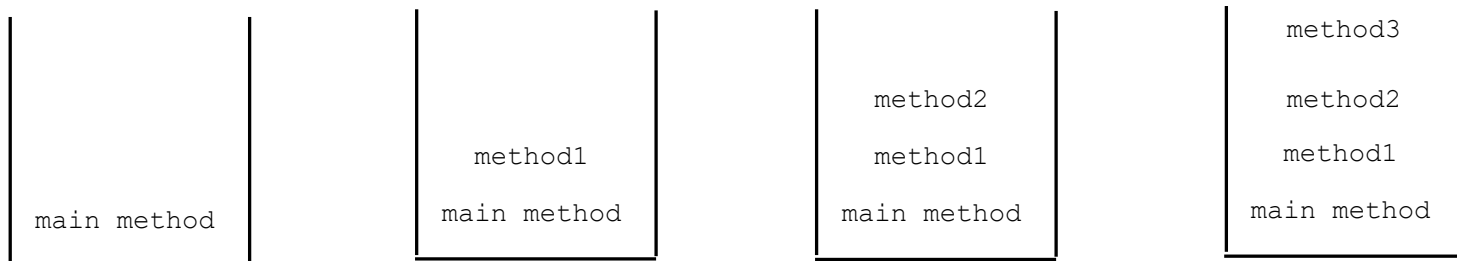
If no handler is found, then the program terminates and prints an error message on the console

Catching exceptions

- The code handling the exception is called the *exception handler*
 - It is found by *propagating the exception* backward through the call stacks, starting from the current method



Call Stack



Checked exceptions

- Remember, the compiler forces the programmer to check and deal with checked exceptions (i.e., any exception other than `Error` or `RuntimeException`)
- If a method declares a checked exception, you must invoke it in a `try-catch` block or declare to throw the exception in the calling method

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

(a)

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

(b)

The `finally` clause

- The `finally` clause is always executed, regardless of whether an exception occurred

```
try {  
    // statements  
}  
catch(TheException ex) {  
    // handling statements  
}  
finally {  
    // final statements  
}
```

Rethrowing exceptions

- Java allows an exception handler to rethrow the exception if the handler cannot process the exception (or simply wants to let its caller be notified of the exception)

```
try {  
    // statements  
}  
catch(TheException ex) {  
    // handling statements before rethrowing  
    throw ex;  
}
```

- You can also throw a new exception along with the original exception
 - This is called *chained exceptions*
 - <https://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html>
 - Liang, section 12.6

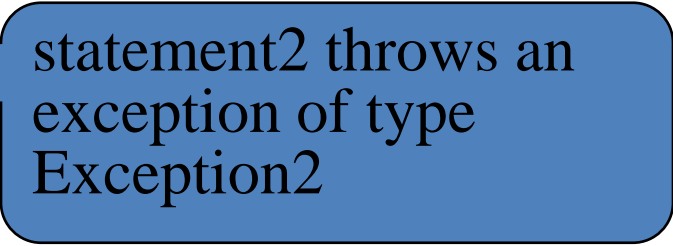
Trace code

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

nextStatement;
```

Trace code

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



statement2 throws an
exception of type
Exception2

Trace code

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



Handling exception

Trace code

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

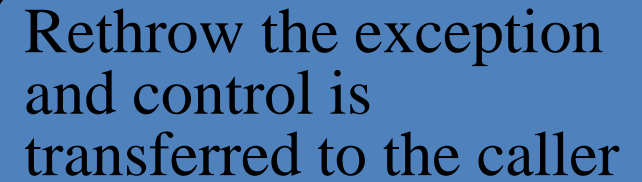


Execute the final block

nextStatement;

Trace code

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



Rethrow the exception
and control is
transferred to the caller

When to use a try-catch block

- Use a try-catch block to deal with **unexpected** error conditions
- Do not use it to deal with simple, **expected** situations

– For example, use this

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

instead of this

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


When to throw exceptions

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

Defining custom exception classes

- Use the exception classes in the Java API whenever possible
- If the predefined classes are insufficient, then you can define a custom exception class by extending the `java.lang.Exception` class

Exception handling

- Exception handling separates error-handling code from normal programming tasks
 - Makes programs easier to read and to modify
- The **try** block contains the code that is executed in **normal** circumstances
- The **catch** block contains the code that is executed in **exceptional** circumstances
- A method should **throw** an exception if the error needs to be handled by its caller
- Warning: 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

Next Lecture

- Assertions
- Text I/O
- Reading
 - Programming with Assertions
 - <https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>
 - Liang
 - Chapter 12