

# Chapter 12 Exception Handling and Text IO



# 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 will introduce in this chapter.



# Objectives

- ☒ To get an overview of exceptions and exception handling (§12.2).
- ☒ To explore the advantages of using exception handling (§12.2).
- ☒ To distinguish exception types: **Error** (fatal) vs. **Exception** (nonfatal) and checked vs. unchecked (§12.3).
- ☒ To declare exceptions in a method header (§12.4.1).
- ☒ To throw exceptions in a method (§12.4.2).
- ☒ To write a **try-catch** block to handle exceptions (§12.4.3).
- ☒ To explain how an exception is propagated (§12.4.3).
- ☒ To obtain information from an exception object (§12.4.4).
- ☒ To develop applications with exception handling (§12.4.5).
- ☒ To use the **finally** clause in a **try-catch** block (§12.5).
- ☒ To use exceptions only for unexpected errors (§12.6).
- ☒ To rethrow exceptions in a **catch** block (§12.7).
- ☒ To create chained exceptions (§12.8).
- ☒ To define custom exception classes (§12.9).



# Exception-Handling Overview

Show runtime error



Quotient

```
System.out.print("Enter two integers: ");
int number1 = input.nextInt();
int number2 = input.nextInt();
System.out.println(number1 + " / " + number2 + " is " +
    (number1 / number2));
```

Fix it using an if statement



QuotientWithIf

```
if (number2 != 0)
    System.out.println(number1 + " / " + number2 + " is " +
        (number1 / number2));
else
    System.out.println("Divisor cannot be zero ");
```

With a method



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;
}
```



# Exception Advantages



QuotientWithException

Run

Now you see the *advantages* of using exception handling. It enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.

```
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) {
    ...
    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 ...");
}
```





# Handling InputMismatchException

InputMismatchExceptionDemo

Run

By handling InputMismatchException, your program will continuously read an input until it is correct.

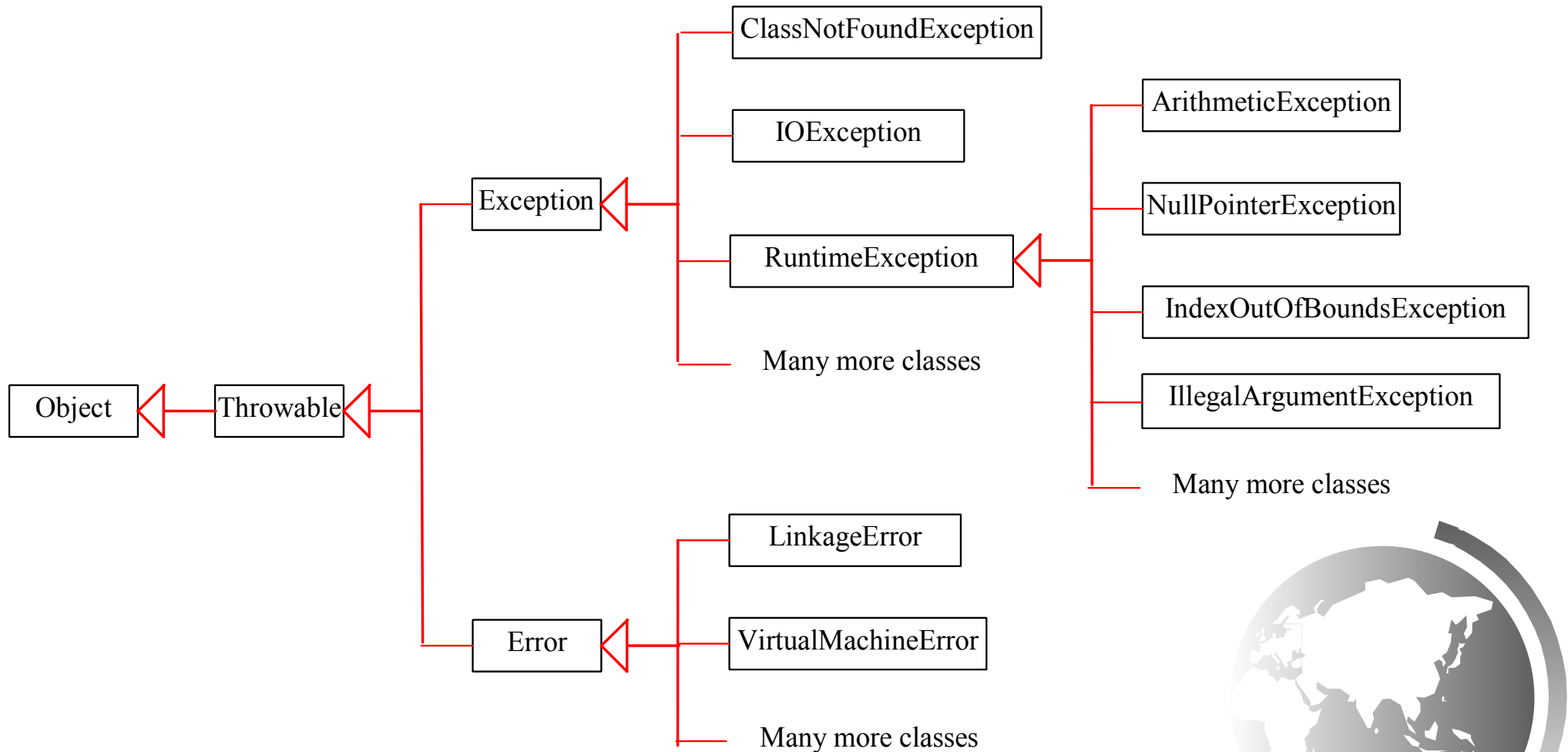
```
boolean continueInput = true;
do {
    try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println(
            "The number entered is " + number);

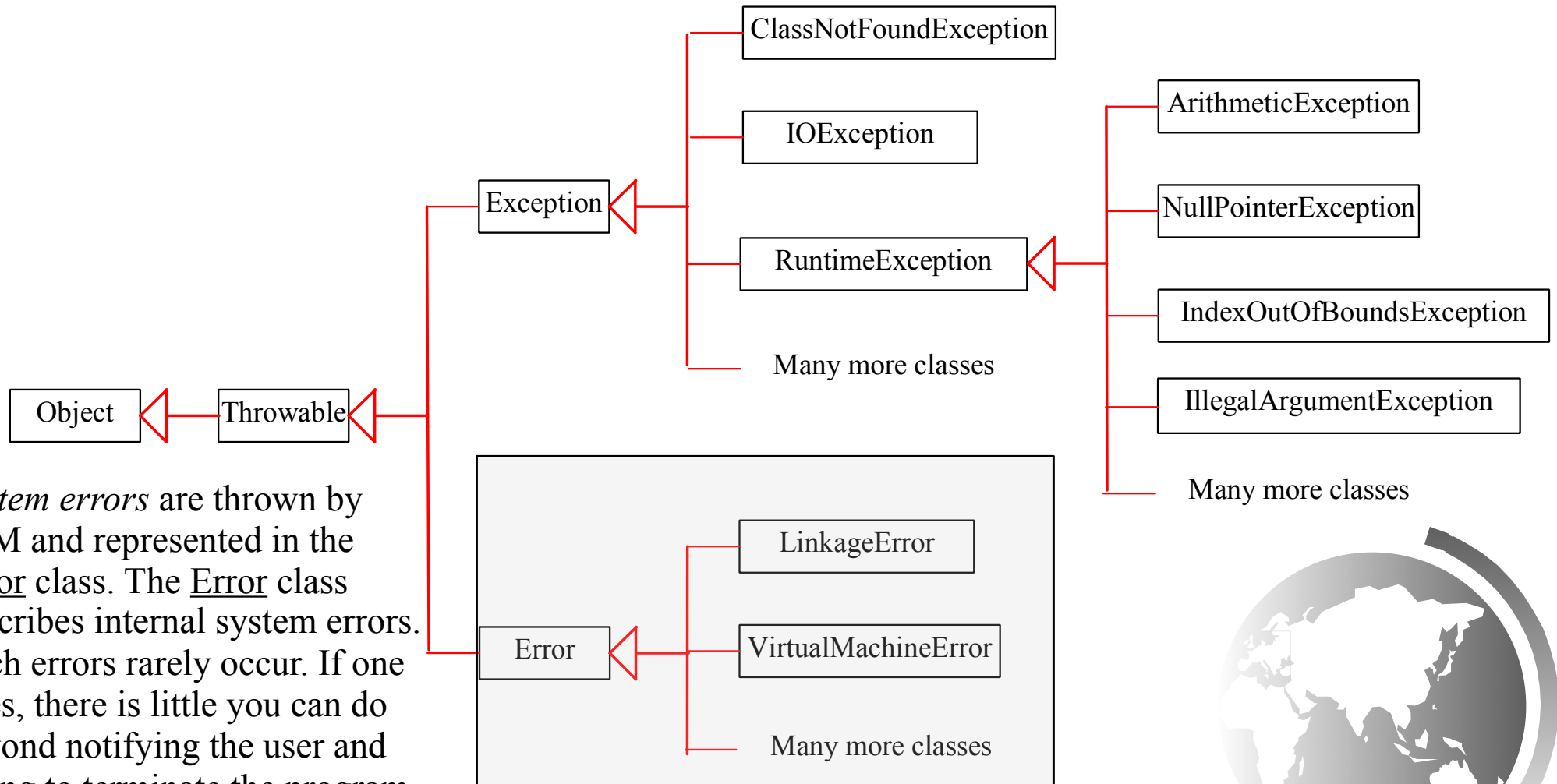
        continueInput = false;
    }
    catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
            "Incorrect input: an integer is required)");
        input.nextLine(); // discard input
    }
} while (continueInput);
```



# Exception Types



# System Errors



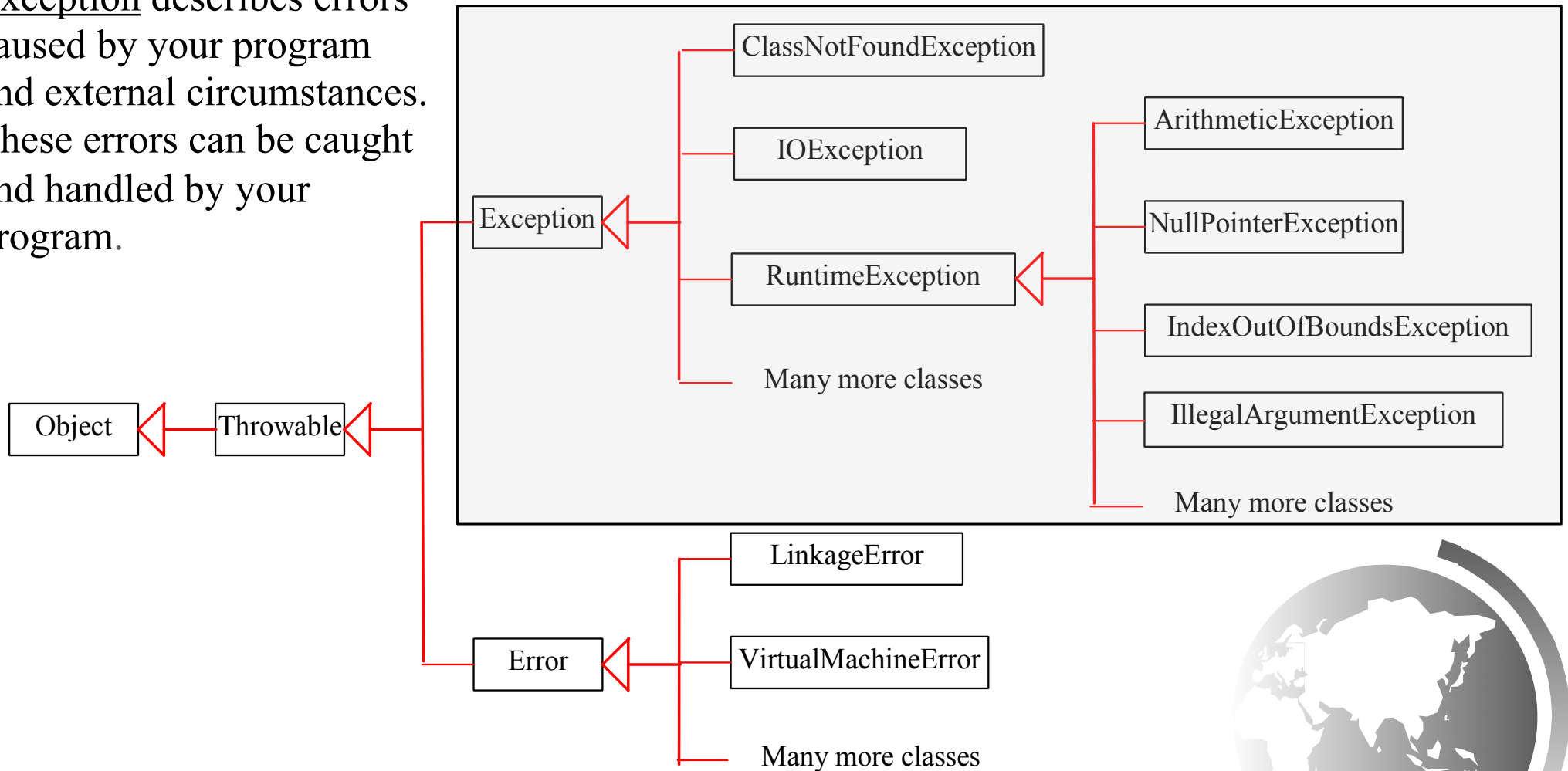
*System errors* are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.





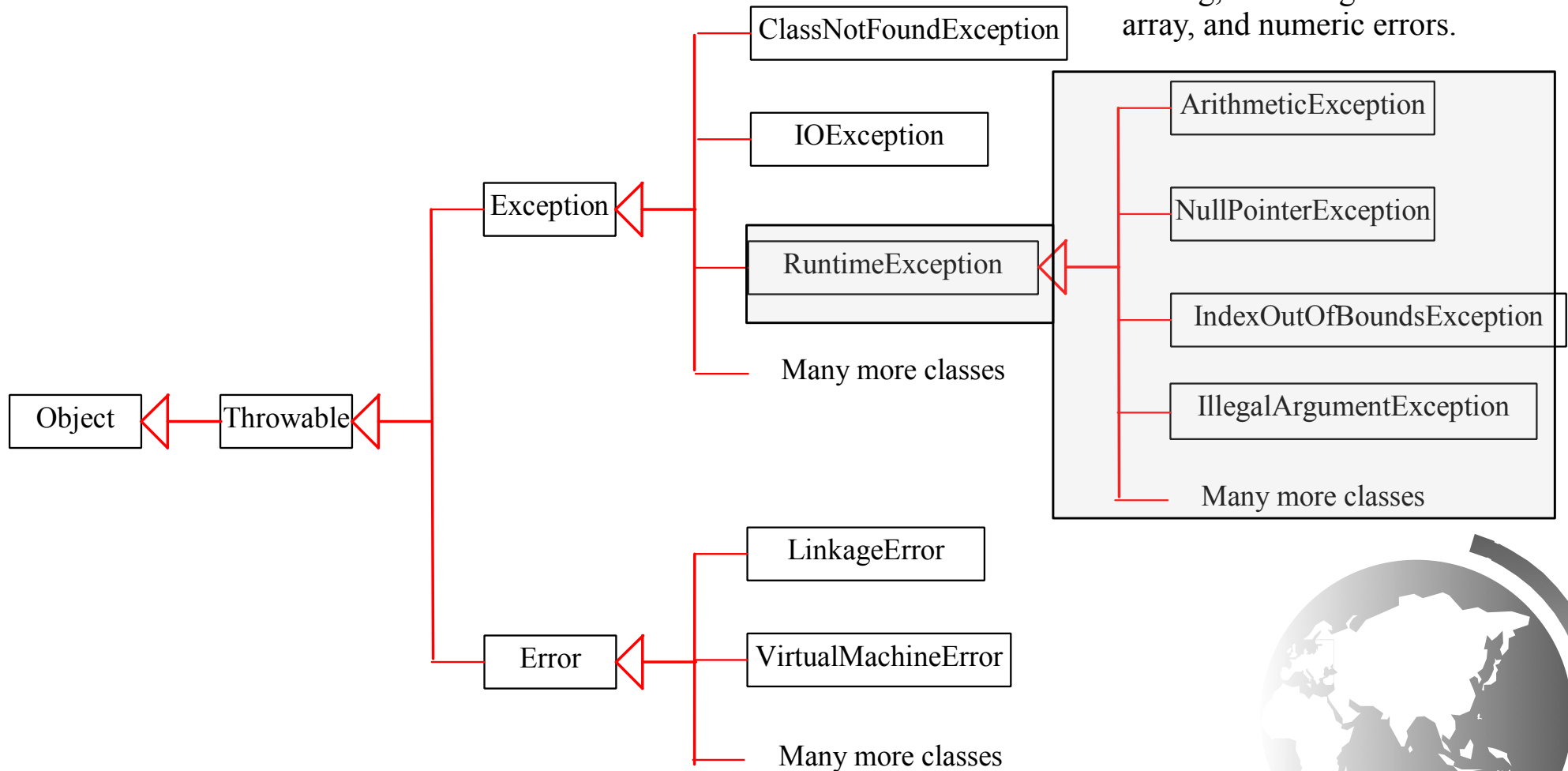
# Exceptions

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

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.

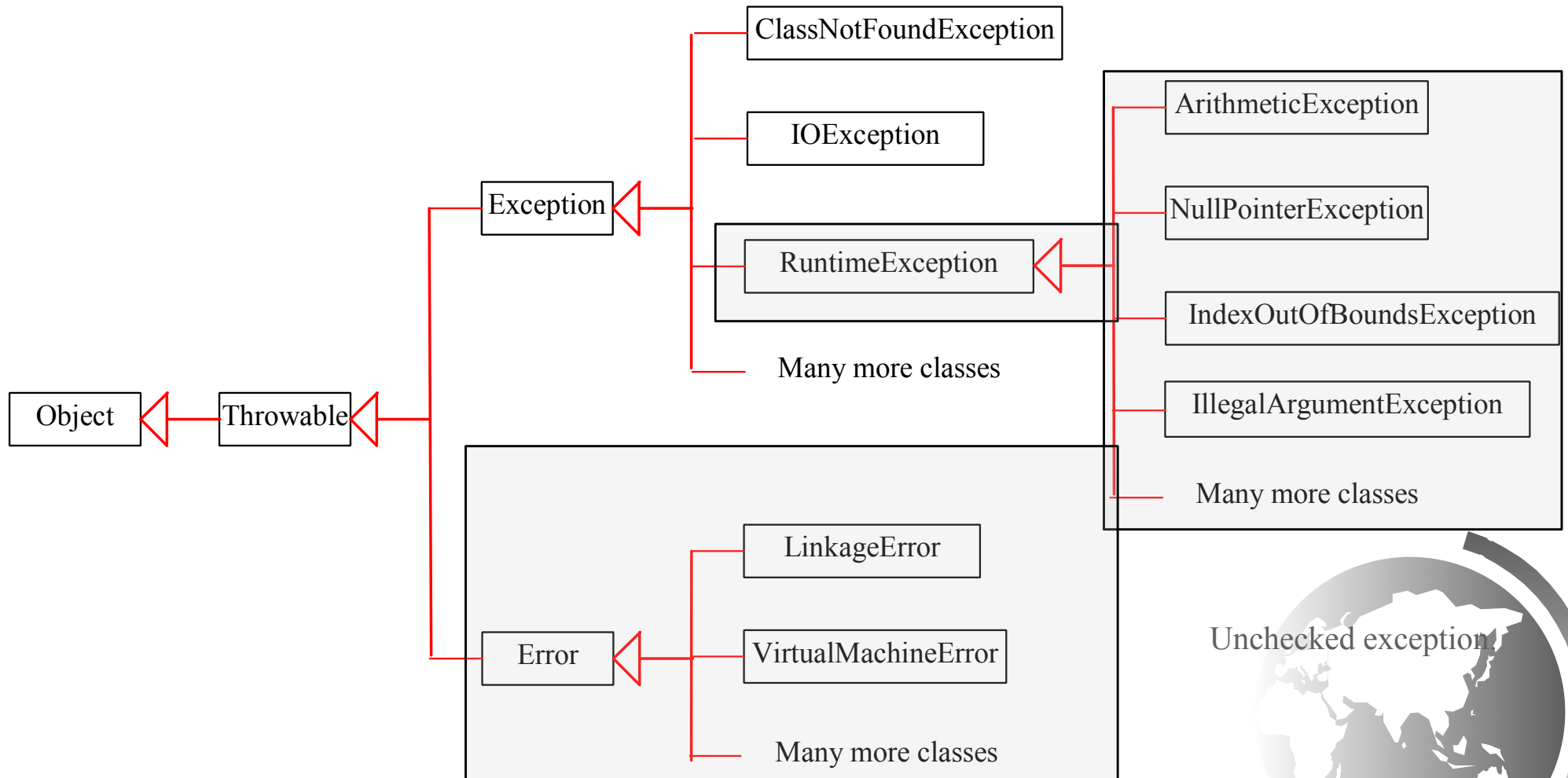


# Unchecked Exceptions

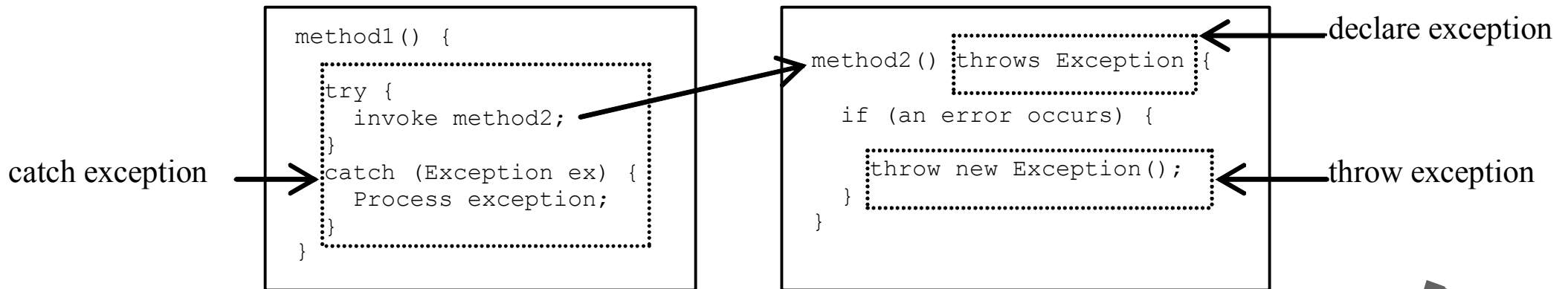
In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in 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.



# 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

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();  
throw ex;
```





# 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");  
}
```

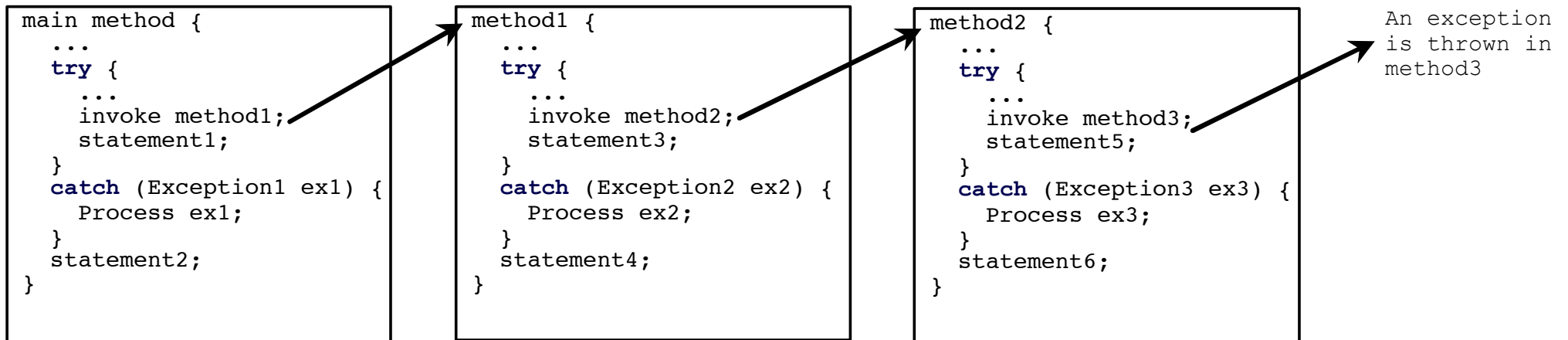


# Catching Exceptions

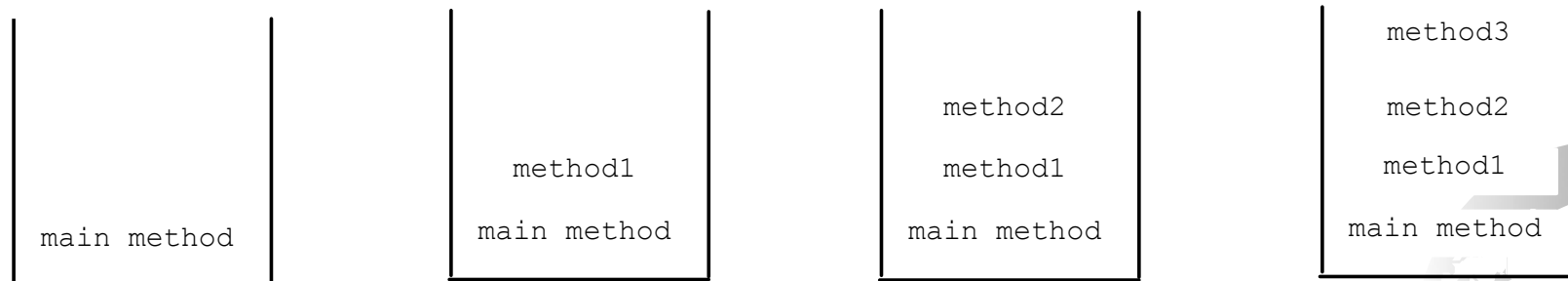
```
try {  
    statements;    // Statements that may throw exceptions  
}  
catch (Exception1 exVar1) {  
    handler for exception1;  
}  
catch (Exception2 exVar2) {  
    handler for exception2;  
}  
...  
catch (ExceptionN exVar3) {  
    handler for exceptionN;  
}
```



# Catching Exceptions



Call Stack



# Catch or Declare Checked Exceptions

Suppose p2 is defined as follows:

```
void p2() throws IOException {  
    if (a file does not exist) {  
        throw new IOException("File does not exist");  
    }  
  
    ...  
}
```

# 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: Declaring, Throwing, and Catching Exceptions



Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 8. The new setRadius method throws an exception if radius is negative.

```
try {  
    CircleWithException c1 = new CircleWithException(5);  
    CircleWithException c2 = new CircleWithException(-5);  
    CircleWithException c3 = new CircleWithException(0);  
}  
catch (IllegalArgumentException ex) {  
    System.out.println(ex);  
}
```

TestCircleWithException

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

CircleWithException

# Rethrowing Exceptions

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



# 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;  
}
```

Suppose an exception  
of type Exception1 is  
thrown in statement2

Next statement;



# 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



# 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




# 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");
```



# Defining Custom Exception Classes

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





# Custom Exception Class Example

In Listing 13.8, 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.



InvalidRadiusException



TestCircleWithCustomException

```
public class InvalidRadiusException extends Exception {  
    private double radius;  
  
    /** Construct an exception */  
    public InvalidRadiusException(double radius) {  
        super("Invalid radius " + radius);  
        this.radius = radius;  
    }  
  
    /** Return the radius */  
    public double getRadius() {  
        return radius;  
    }  
}
```

# Assertions

An assertion is a Java statement that enables you to assert an assumption about your program. An assertion contains a Boolean expression that should be true during program execution. Assertions can be used to assure program correctness and avoid logic errors.



# Declaring Assertions

An *assertion* is declared using the new Java keyword assert in JDK 1.4 as follows:

*assert assertion;* or  
*assert assertion : detailMessage;*

where **assertion** is a Boolean expression and *detailMessage* is a primitive-type or an Object value.



# Executing Assertions

When an assertion statement is executed, Java evaluates the assertion. If it is false, an `AssertionError` will be thrown. The `AssertionError` class has a no-arg constructor and seven overloaded single-argument constructors of type `int`, `long`, `float`, `double`, `boolean`, `char`, and `Object`.

For the first `assert` statement with no detail message, the no-arg constructor of `AssertionError` is used. For the second `assert` statement with a detail message, an appropriate `AssertionError` constructor is used to match the data type of the message. Since `AssertionError` is a subclass of `Error`, when an assertion becomes false, the program displays a message on the console and exits.

# Executing Assertions Example

```
public class AssertionDemo {  
    public static void main(String[] args) {  
        int i; int sum = 0;  
        for (i = 0; i < 10; i++) {  
            sum += i;  
        }  
        assert i == 10;  
        assert sum > 10 && sum < 5 * 10 : "sum is " + sum;  
    }  
}
```



# Compiling Programs with Assertions

Since assert is a new Java keyword introduced in JDK 1.4, you have to compile the program using a JDK 1.4 compiler. Furthermore, you need to include the switch **–source 1.4** in the compiler command as follows:

**javac –source 1.4 AssertionDemo.java**

NOTE: If you use JDK 1.5, there is no need to use the **–source 1.4** option in the command.



# Running Programs with Assertions

By default, the assertions are disabled at runtime. To enable it, use the switch **—enableassertions**, or **—ea** for short, as follows:

**java —ea AssertionDemo**

Assertions can be selectively enabled or disabled at class level or package level. The disable switch is **—disableassertions** or **—da** for short. For example, the following command enables assertions in package package1 and disables assertions in class Class1.

**java —ea:package1 —da:Class1 AssertionDemo**

# Using Exception Handling or Assertions

Assertion should not be used to replace exception handling. Exception handling deals with unusual circumstances during program execution. Assertions are to assure the correctness of the program. Exception handling addresses robustness and assertion addresses correctness. Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks. Assertions are checked at runtime and can be turned on or off at startup time.





## Using Exception Handling or Assertions, cont.

*Do not use assertions for argument checking in public methods.* Valid arguments that may be passed to a public method are considered to be part of the method's contract. The contract must always be obeyed whether assertions are enabled or disabled. For example, the following code in the Circle class should be rewritten using exception handling.

```
public void setRadius(double newRadius) {  
    assert newRadius >= 0;  
    radius = newRadius;  
}
```



# Using Exception Handling or Assertions, cont.

*Use assertions to reaffirm assumptions.* This gives you more confidence to assure correctness of the program. A common use of assertions is to replace assumptions with assertions in the code.



# Using Exception Handling or Assertions, cont.

Another good use of assertions is place assertions in a switch statement without a default case. For example,

```
switch (month) {  
    case 1: ... ; break;  
    case 2: ... ; break;  
    ...  
    case 12: ... ; break;  
    default: assert false : "Invalid month: " + month  
}
```

