Exceptions and Assertions

Syntax, Runtime, and Logic Errors

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

✧ *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

```java
import java.util.Scanner;

public class ExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int number = scanner.nextInt();

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

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

```java
import java.util.*;

public class HandleExceptionDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        boolean continueInput = true;

        do {
            try {
                System.out.print("Enter an integer: ");
                int number = scanner.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)");
                scanner.nextLine(); // discard input
            }
        } while (continueInput);
    }
}
```
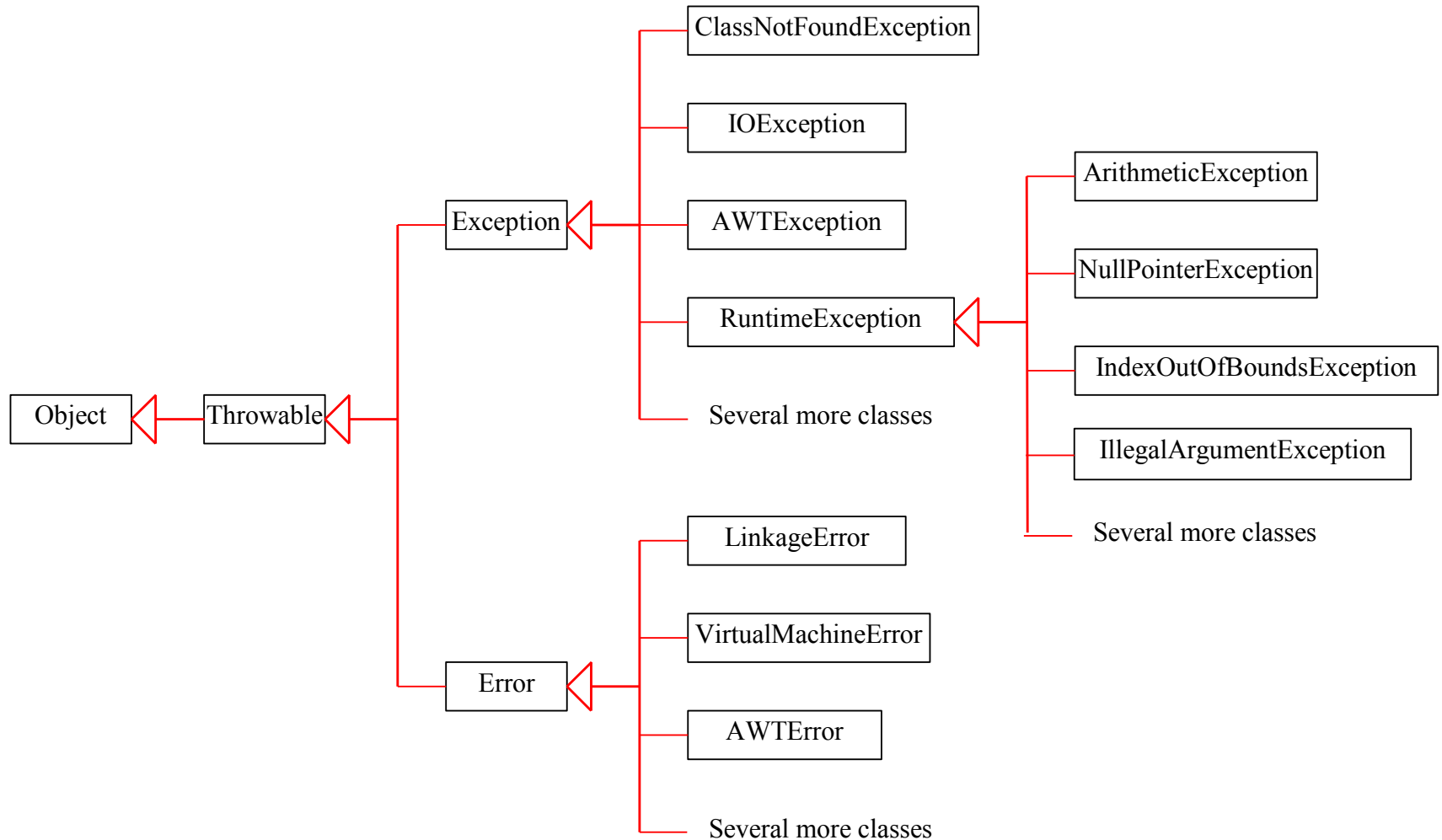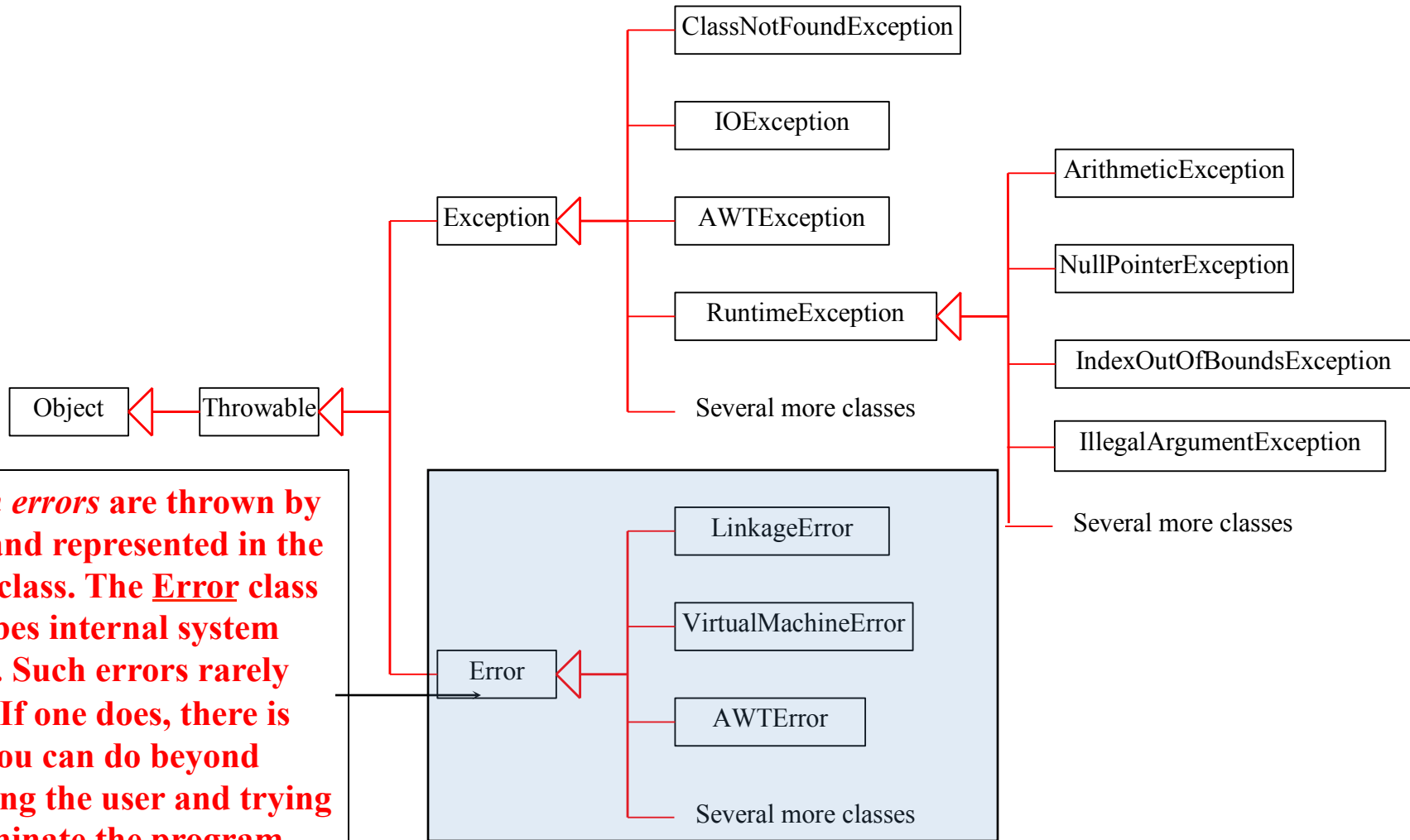
If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

Exception Classes

ClassNotFoundException

IOException

AWTException

Exception

RuntimeException

Several more classes

ArithmeticException

NullPointerException

IndexOutOfBoundsException

IllegalArgumentException

Several more classes

Object

Throwable

LinkageError

VirtualMachineError

Error

AWTError

Several more classes

System Errors

ClassNotFoundException

IOException

AWTException

Exception

RuntimeException

Several more classes

ArithmeticException

NullPointerException

IndexOutOfBoundsException
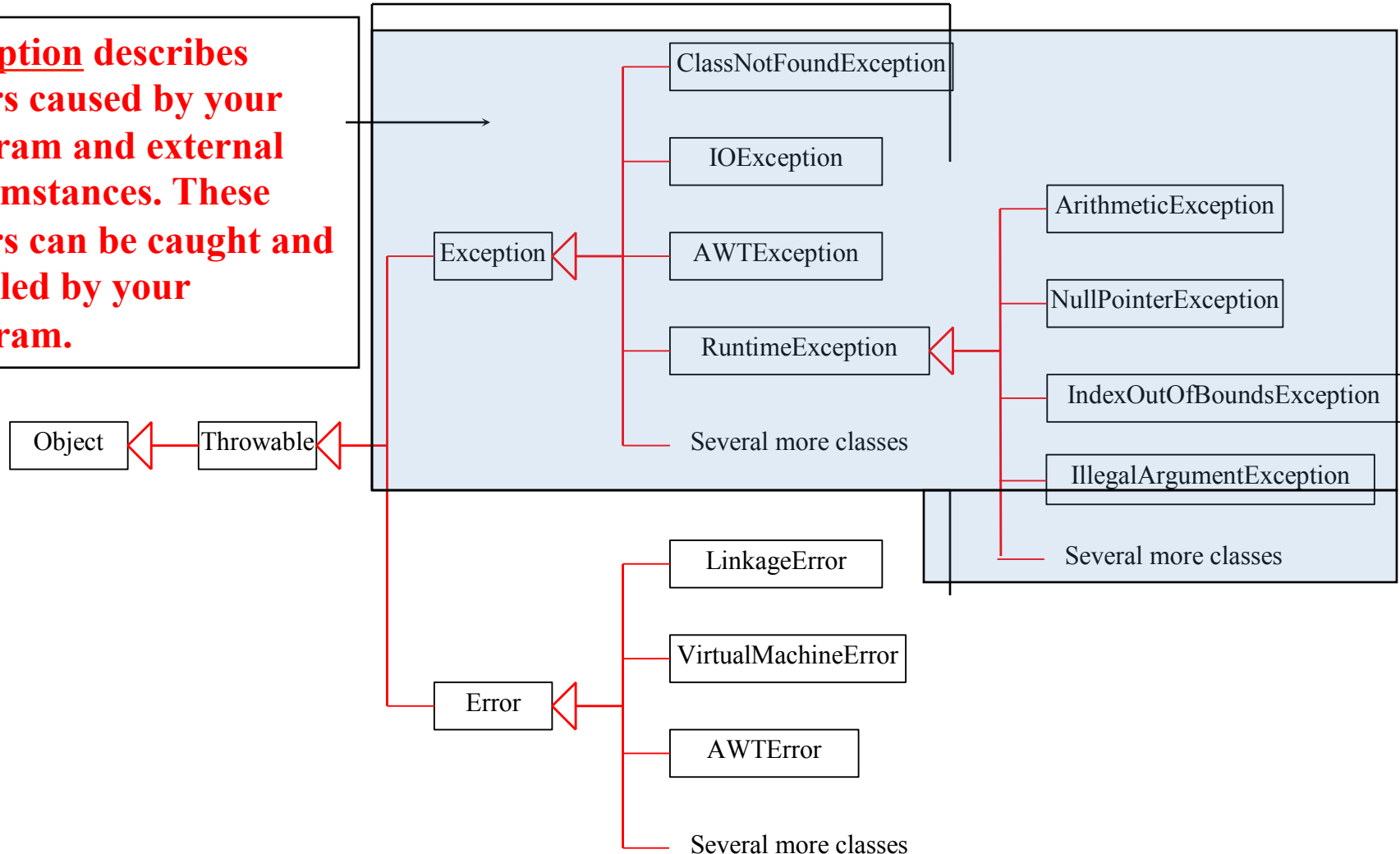
IllegalArgumentException

Several more classes

Object

Throwable

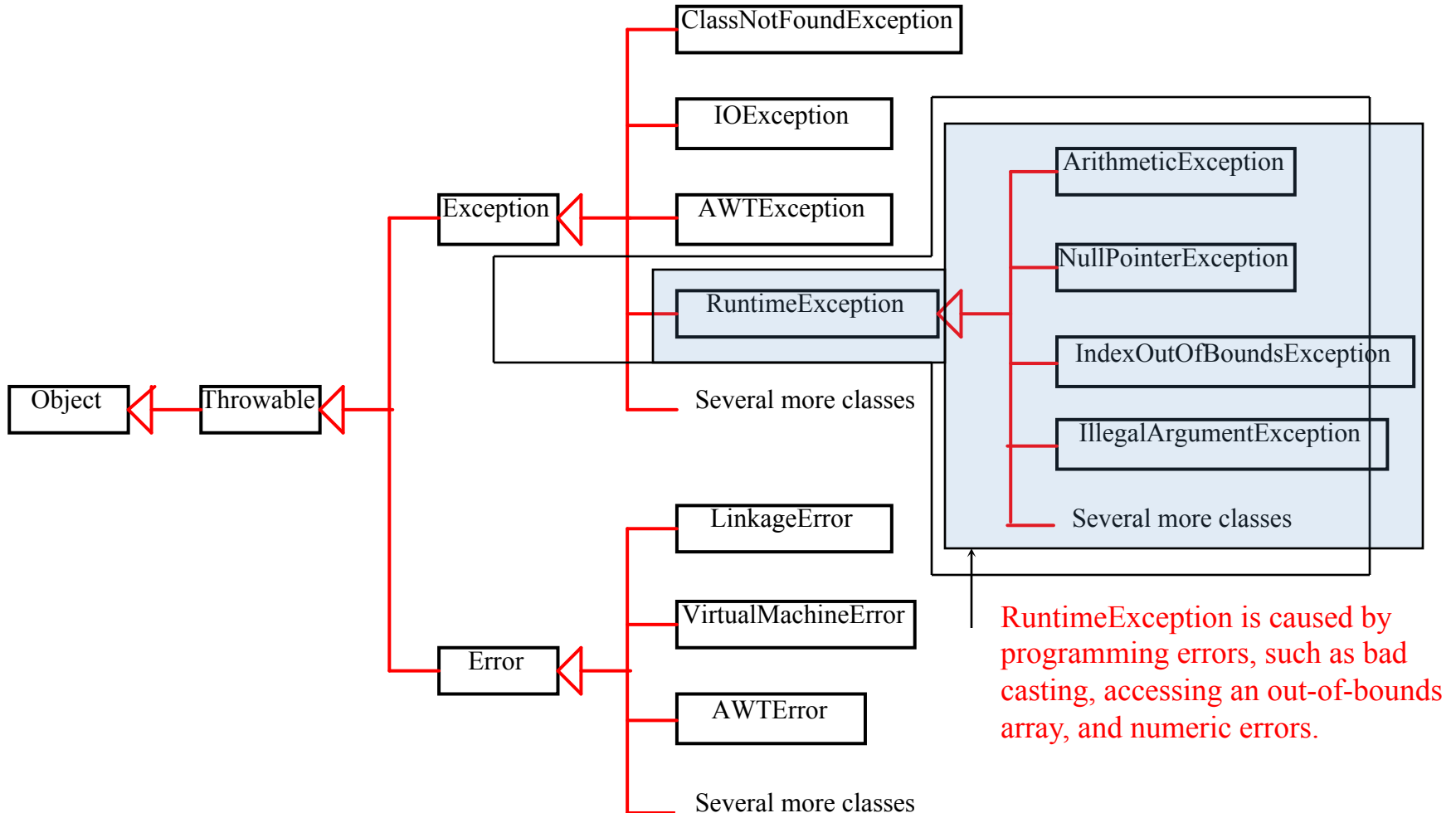*System errors* **are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.**

Error

LinkageError

VirtualMachineError

AWTError

Several more classes

Exceptions

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

ClassNotFoundException
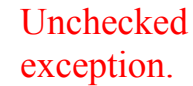
IOException

ArithmeticException

AWTException

Exception

NullPointerException

RuntimeException

IndexOutOfBoundsException

Object

Throwable

IllegalArgumentException

Several more classes

Several more classes

LinkageError

VirtualMachineError

Error

AWTError

Several more classes

Runtime Exceptions

ClassNotFoundException

IOException

AWTException

Exception

RuntimeException

Several more classes

ArithmeticException

NullPointerException

IndexOutOfBoundsException

IllegalArgumentException

Several more classes

Object

Throwable

LinkageError

VirtualMachineError

Error

AWTError

Several more classes

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.
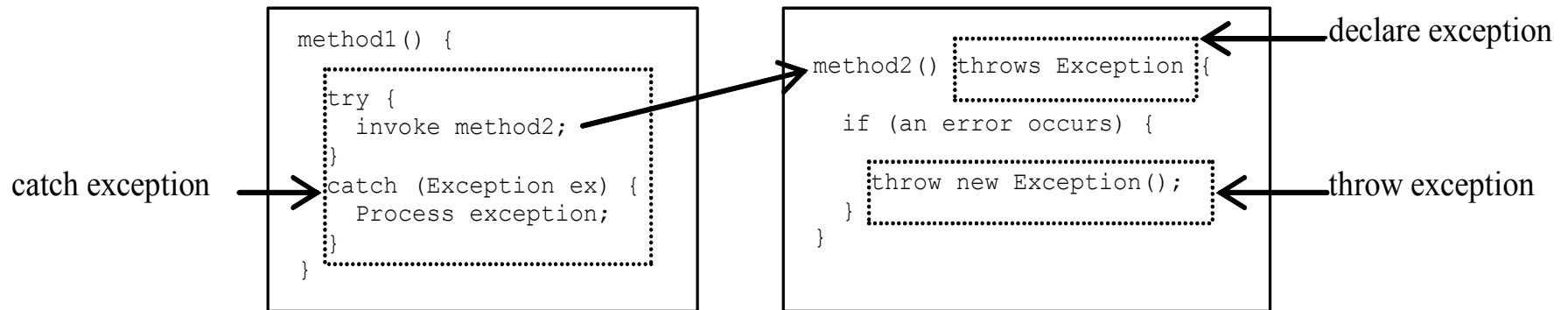
Unchecked Exceptions

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.

Checked or Unchecked Exceptions

ClassNotFoundException

IOException

AWTException

Exception

ArithmeticException

NullPointerException

RuntimeException

IndexOutOfBoundsException

IllegalArgumentException

Several more classes

Object

Throwable

Several more classes

LinkageError

VirtualMachineError

Error

AWTError

Unchecked exception.

Several more classes

Declaring, Throwing, and Catching Exceptions

```
method1() {

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

}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

catch exception

declare exception

throw 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

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

```
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;
}
```

An exception is thrown in method3

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: Exceptions in GUI Applications

- An error message appears on the console, but the GUI application continues running.

- Write a program that creates a user interface to perform integer divisions. The user enters two numbers in the text fields Number 1 and Number 2. The division of Number 1 and Number 2 is displayed in the Result field when the Divide button is clicked.

Integer Division
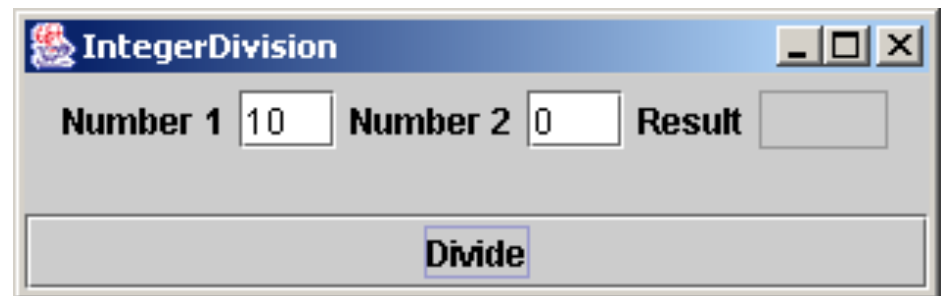
Rethrowing Exceptions

```
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;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2
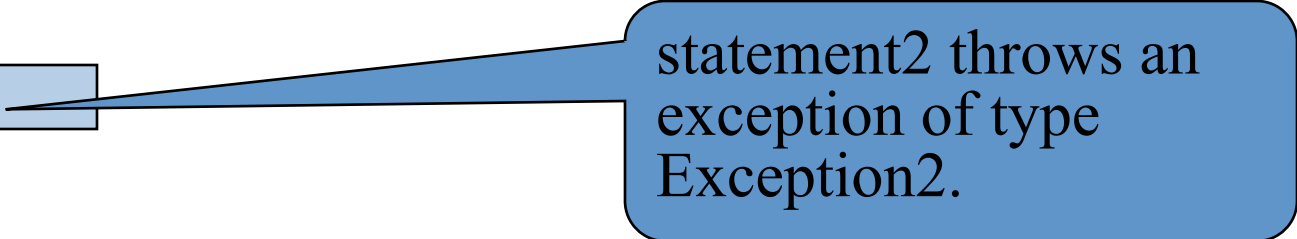
Trace a Program Execution
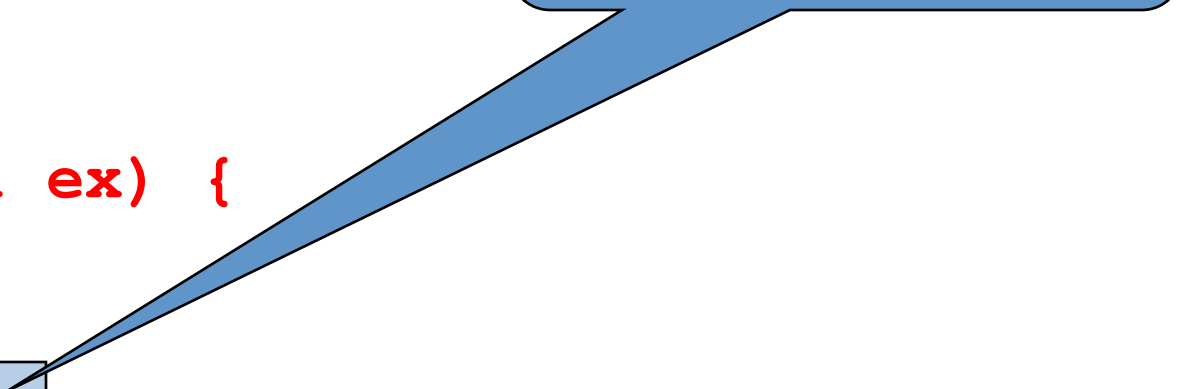
```
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}

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;
```
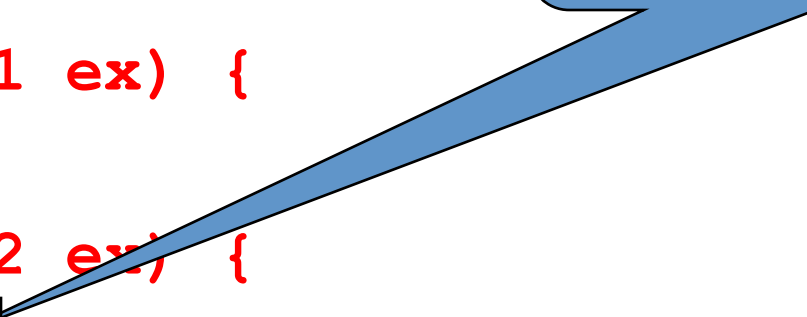
Execute the final block

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

```java
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.

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 a**ssert**ion 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 <u>assert</u> 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

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 should be rewritten using exception handling as shown in Lines 28-35 in Circle.java in Listing 17.1.

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

Using Exception Handling or Assertions

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

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
}
```