

# Chapter 1

## Exception Handling and Text IO



# Exception Handling



# 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?

# Objectives

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

# Exception-Handling Overview – Show runtime error

```
1  import java.util.Scanner;
2
3  public class Quotient {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6
7          // Prompt the user to enter two integers
8          System.out.print("Enter two integers: ");
9          int number1 = input.nextInt();
10         int number2 = input.nextInt();
11
12         System.out.println(number1 + " / " + number2 + " is " +
13             (number1 / number2));
14     }
15 }
```

Enter two integers: 5 2   
5 / 2 is 2

Enter two integers: 3 0   
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Quotient.main(Quotient.java:11)

# Exception-Handling Overview – Fix it using if statement

```
1  import java.util.Scanner;
2
3  public class QuotientWithIf {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6
7          // Prompt the user to enter two integers
8          System.out.print("Enter two integers: ");
9          int number1 = input.nextInt();
10         int number2 = input.nextInt();
11
12         if (number2 != 0)
13             System.out.println(number1 + " / " + number2
14                                 + " is " + (number1 / number2));
15         else
16             System.out.println("Divisor cannot be zero ");
17     }
18 }
```

Enter two integers: 5 0   
Divisor cannot be zero

# Exception-Handling Overview – with a method

```
1  import java.util.Scanner;
2
3  public class QuotientWithMethod {
4      public static int quotient(int number1, int number2) {
5          if (number2 == 0) {
6              System.out.println("Divisor cannot be zero");
7              System.exit(1);
8          }
9
10         return number1 / number2;
11     }
12
13     public static void main(String[] args) {
14         Scanner input = new Scanner(System.in);
15
16         // Prompt the user to enter two integers
17         System.out.print("Enter two integers: ");
18         int number1 = input.nextInt();
19         int number2 = input.nextInt();
20
21         int result = quotient(number1, number2);
22         System.out.println(number1 + " / " + number2 + " is "
23             + result);
24     }
25 }
```

Enter two integers: 5 3   
5 / 3 is 1

Enter two integers: 5 0   
Divisor cannot be zero

# Exception Advantages

```
1 import java.util.Scanner;
2
3 public class QuotientWithException {
4     public static int quotient(int number1, int number2) {
5         if (number2 == 0)
6             throw new ArithmeticException("Divisor cannot be zero");
7
8         return number1 / number2;
9     }
10
11     public static void main(String[] args) {
12         Scanner input = new Scanner(System.in);
13
14         // Prompt the user to enter two integers
15         System.out.print("Enter two integers: ");
16         int number1 = input.nextInt();
17         int number2 = input.nextInt();
18
19         try {
20             int result = quotient(number1, number2);
21             System.out.println(number1 + " / " + number2 + " is "
22                 + result);
23         }
24         catch (ArithmeticException ex) {
25             System.out.println("Exception: an integer " +
26                 "cannot be divided by zero ");
27         }
28
29         System.out.println("Execution continues ...");
30     }
31 }
```

Enter two integers: 5 3   
5 / 3 is 1  
Execution continues ...

Enter two integers: 5 0   
Exception: an integer cannot be divided by zero  
Execution continues ...

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.



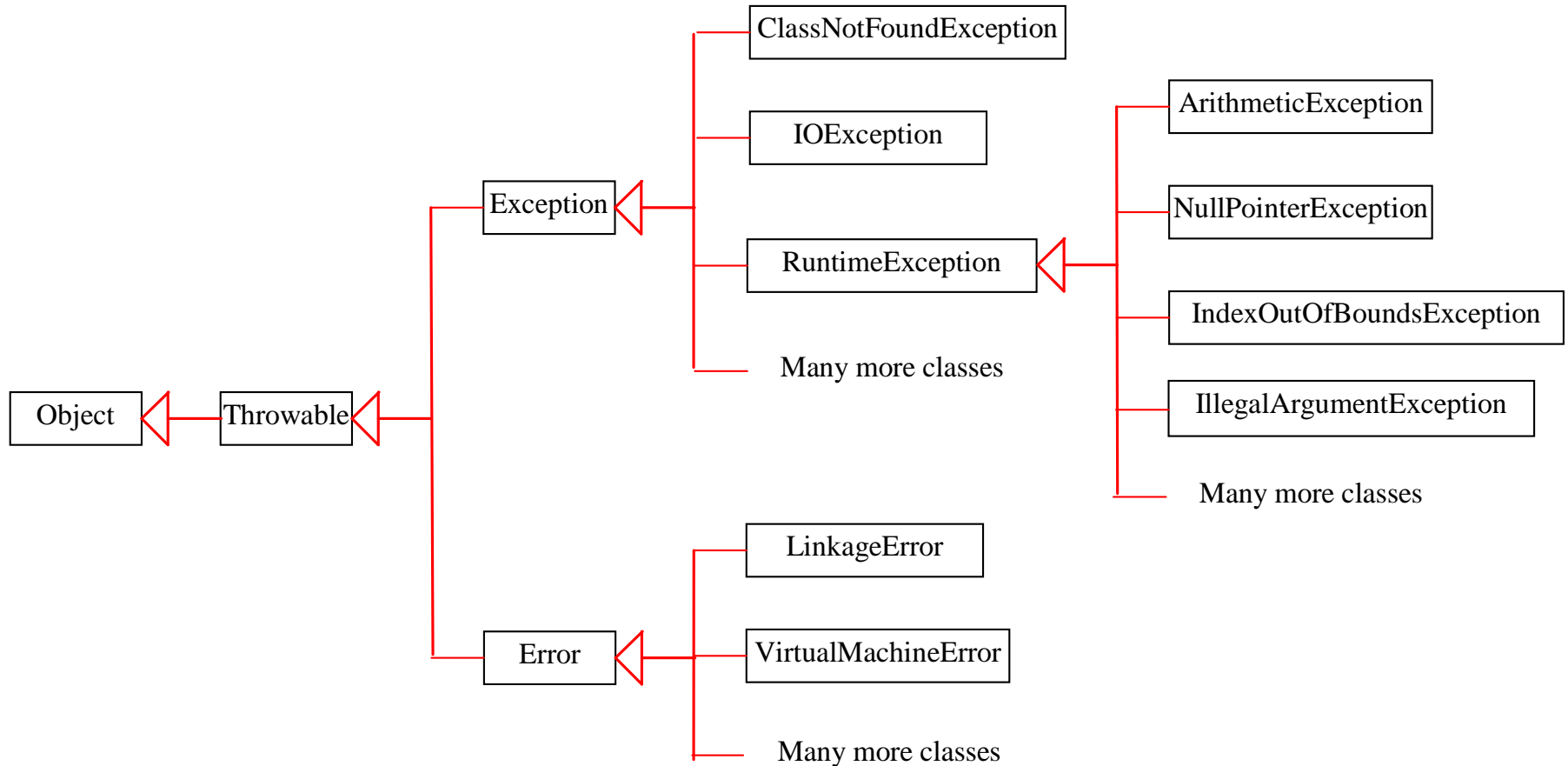
# Handling InputMismatchException

```
1  import java.util.*;
2
3  public class InputMismatchExceptionDemo {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6          boolean continueInput = true;
7
8          do {
9              try {
10                 System.out.print("Enter an integer: ");
11                 int number = input.nextInt();
12                 // If an InputMismatchException occurs
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                 input.nextLine(); // Discard input
23             }
24         } while (continueInput);
25     }
26 }
```

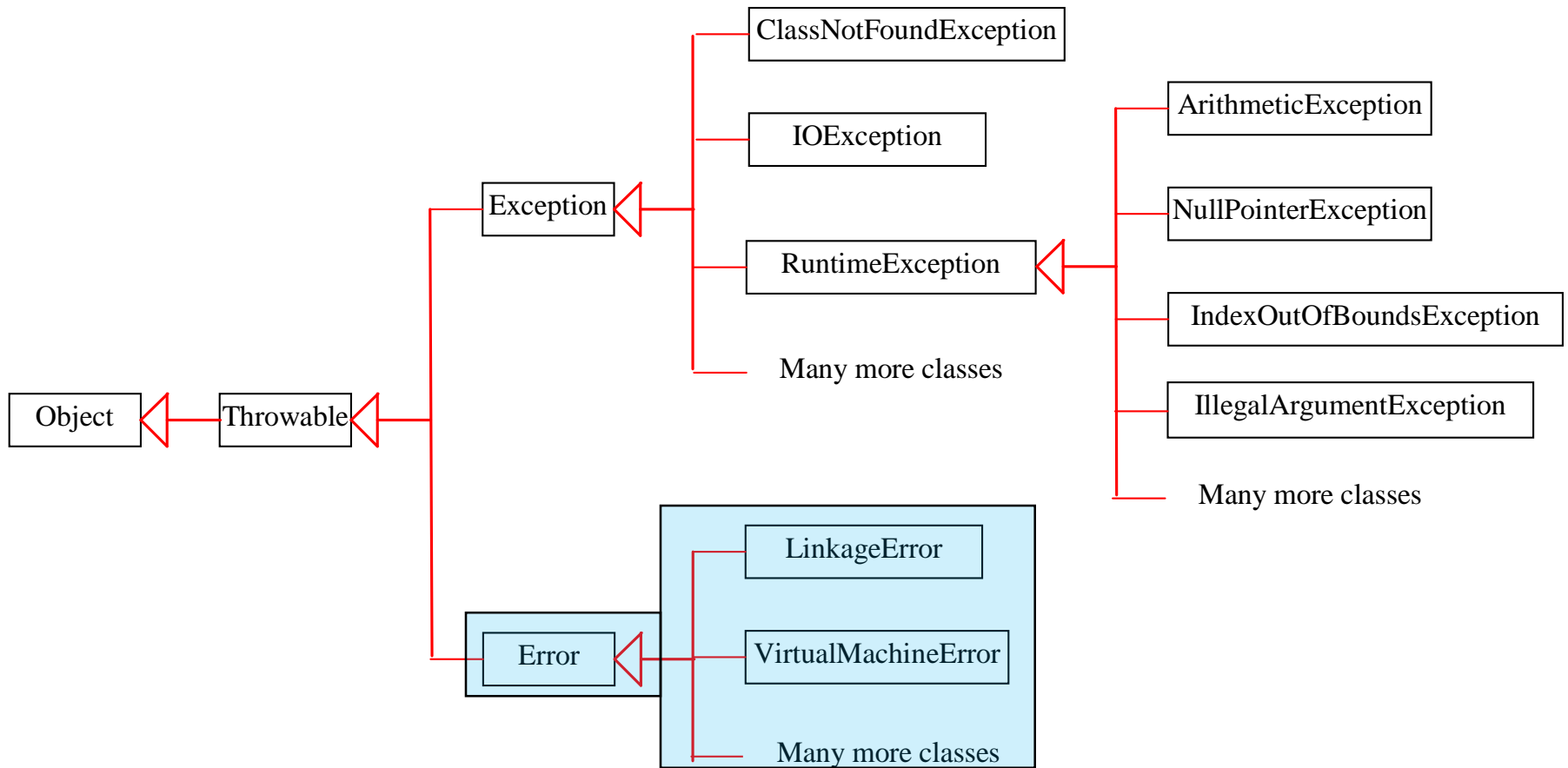
By handling  
InputMismatchException,  
your program will  
continuously read an input  
until it is correct.

```
Enter an integer: 3.5 ↵ Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 ↵ Enter
The number entered is 4
```

# Exception Types

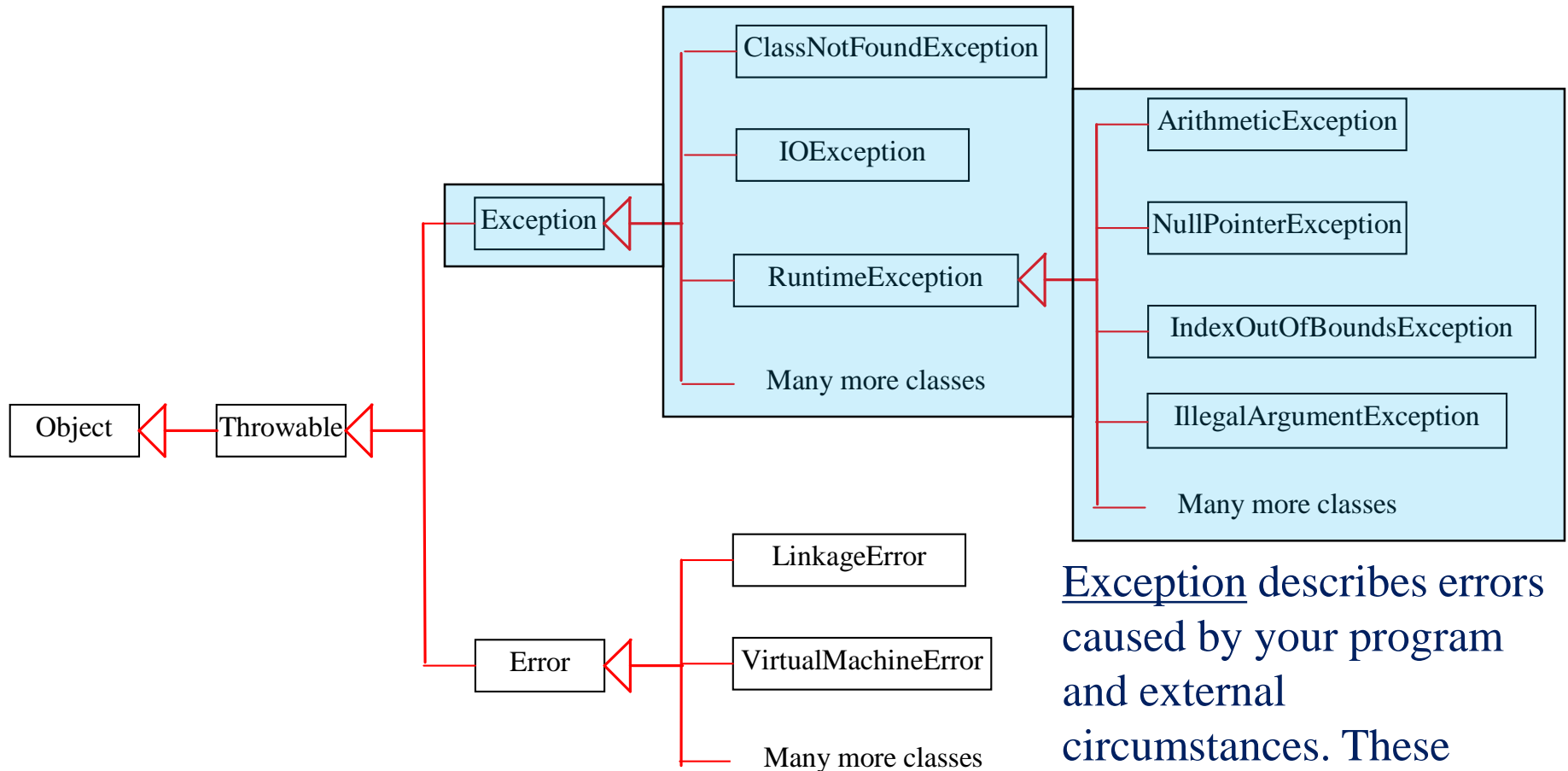


# System Errors



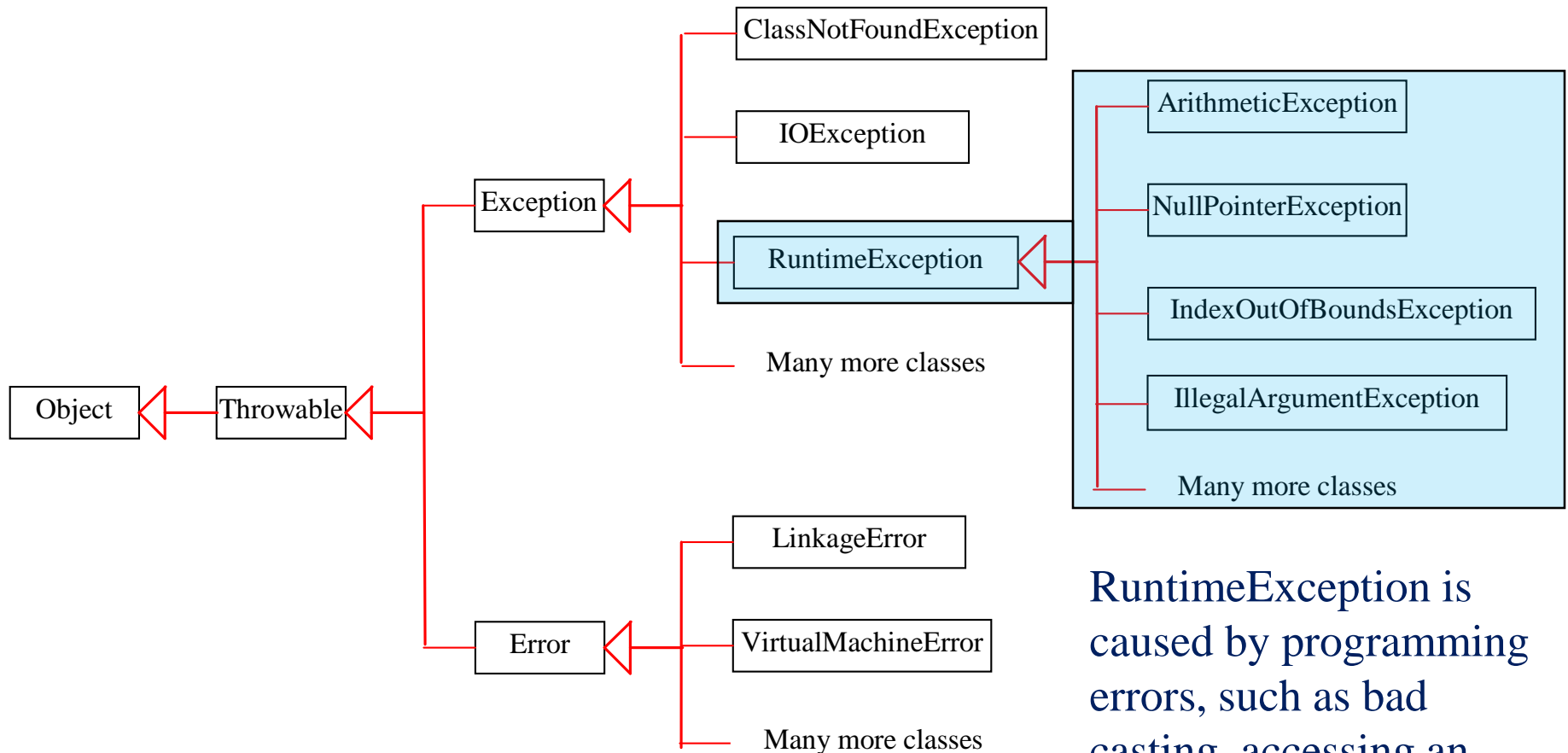
*System errors* are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

# 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.

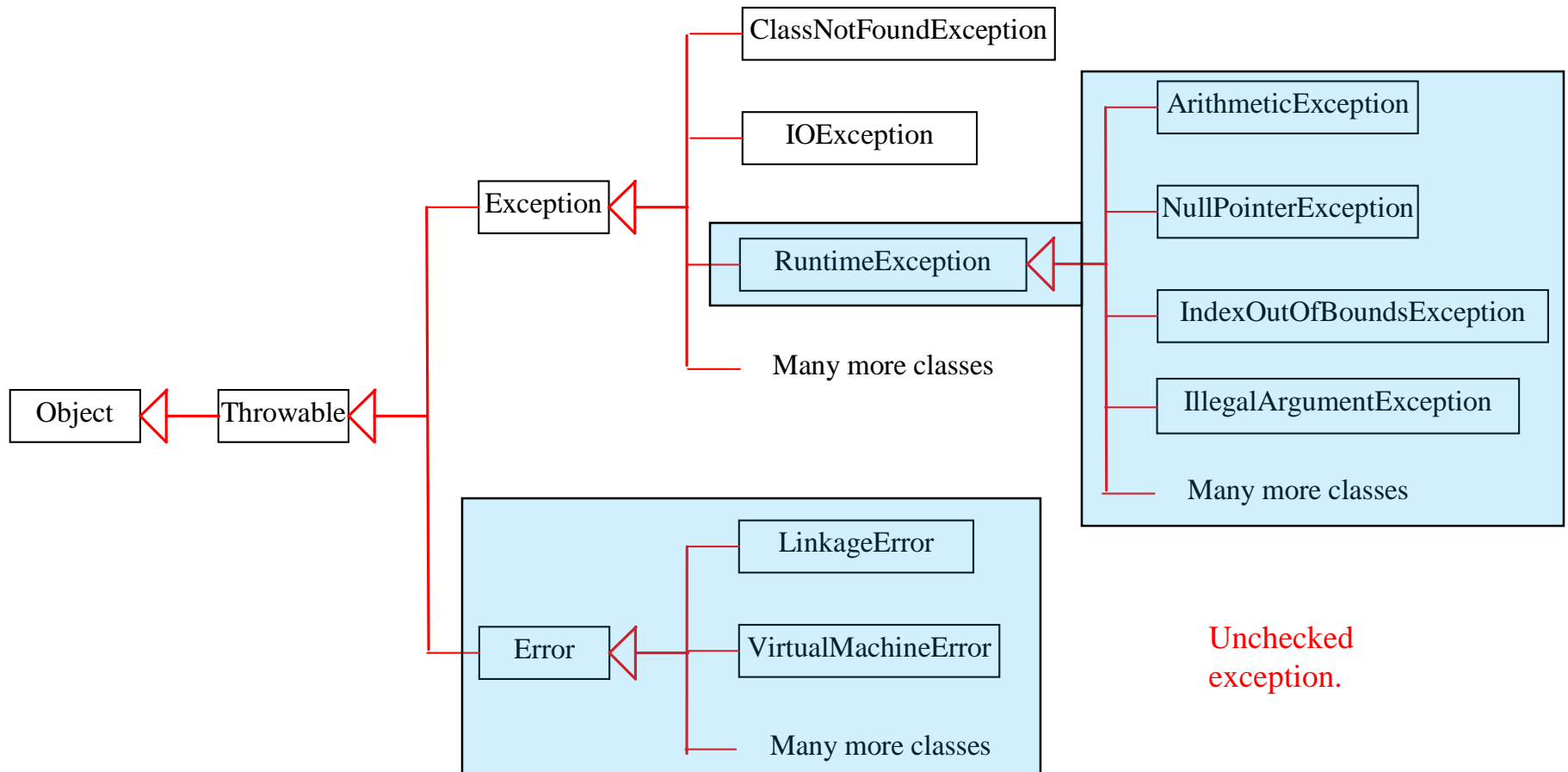
# Checked Exceptions vs. Unchecked Exceptions

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

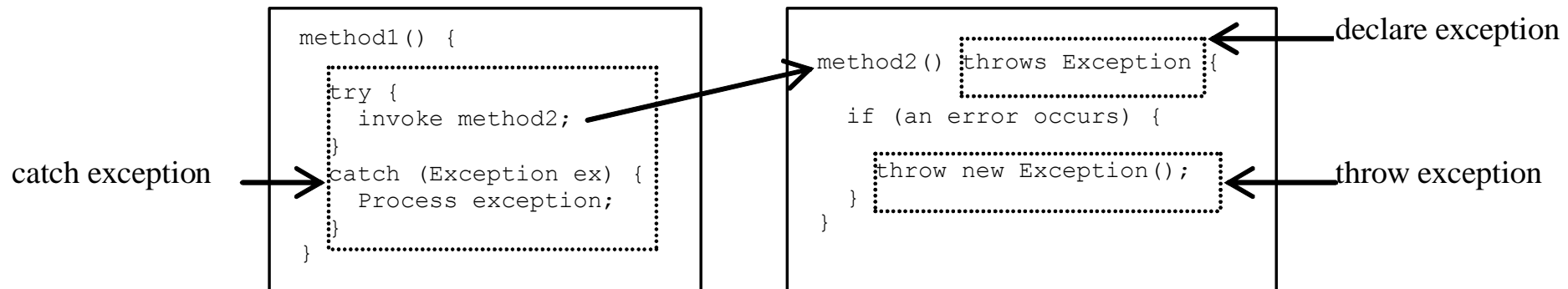
In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in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

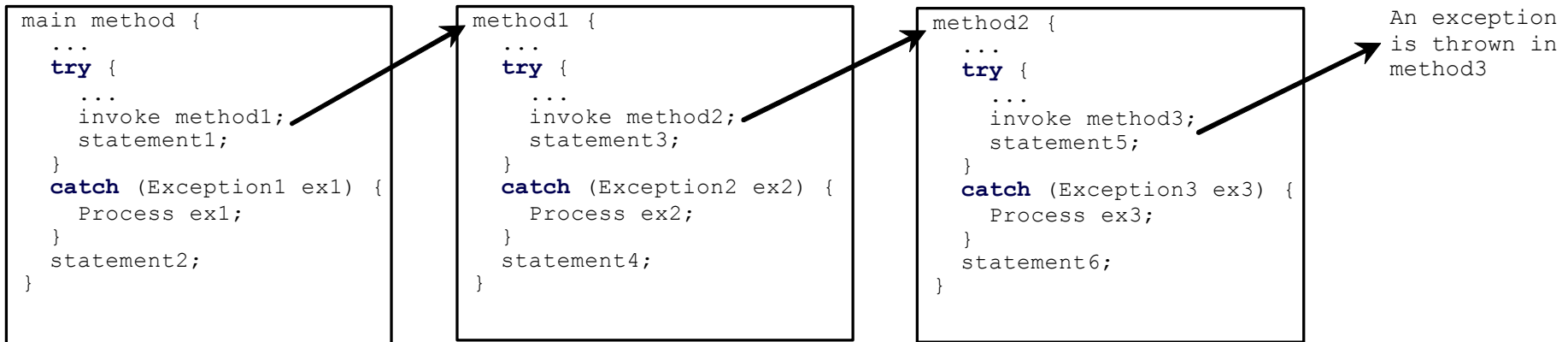
# Throwing Exceptions Example

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

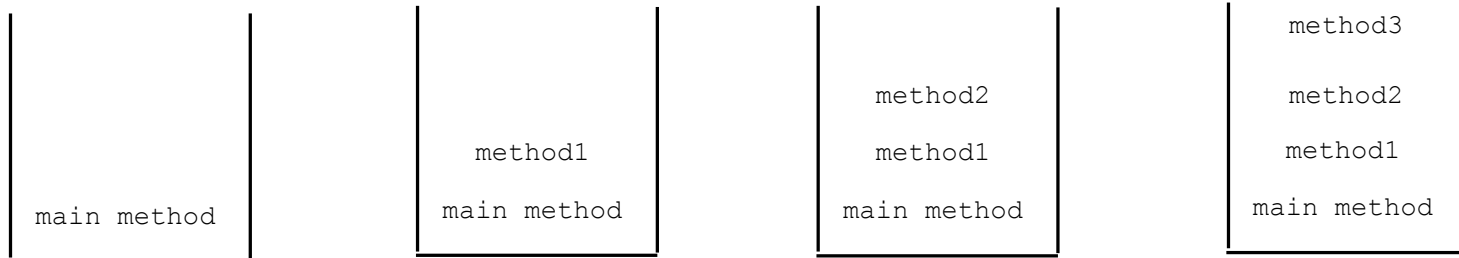
# Catching Exceptions

```
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. The new setRadius method throws an exception if radius is negative.

# Example: Declaring, Throwing, and Catching Exceptions

```
1 public class CircleWithException {
2     /** The radius of the circle */
3     private double radius;
4
5     /** The number of the objects created */
6     private static int numberOfObjects = 0;
7
8     /** Construct a circle with radius 1 */
9     public CircleWithException() {
10         this(1.0);
11     }
12
13     /** Construct a circle with a specified radius */
14     public CircleWithException(double newRadius) {
15         setRadius(newRadius);
16         numberOfObjects++;
17     }
18
19     /** Return radius */
20     public double getRadius() {
21         return radius;
22     }
23 }
```

```
24     /** Set a new radius */
25     public void setRadius(double newRadius)
26         throws IllegalArgumentException {
27         if (newRadius >= 0)
28             radius = newRadius;
29         else
30             throw new IllegalArgumentException(
31                 "Radius cannot be negative");
32     }
33
34     /** Return numberOfObjects */
35     public static int getNumberOfObjects() {
36         return numberOfObjects;
37     }
38
39     /** Return the area of this circle */
40     public double findArea() {
41         return radius * radius * 3.14159;
42     }
43 }
```

# Example: Declaring, Throwing, and Catching Exceptions

```
1 public class TestCircleWithException {
2     public static void main(String[] args) {
3         try {
4             CircleWithException c1 = new CircleWithException(5);
5             CircleWithException c2 = new CircleWithException(-5);
6             CircleWithException c3 = new CircleWithException(0);
7         }
8         catch (IllegalArgumentException ex) {
9             System.out.println(ex);
10        }
11
12        System.out.println("Number of objects created: " +
13            CircleWithException.getNumberOfObjects());
14    }
15 }
```

```
java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1
```

# 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;  
}
```

The final block is  
always executed

Next statement;

# 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;  
}
```

The exception is handled.

Next statement;

# Trace a Program Execution

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

The final block is  
always executed.

Next statement;

# 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;  
}
```

statement2 throws an exception of type Exception2.

Next statement;

# Trace a Program Execution

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



Handling exception

Next statement;

# Trace a Program Execution

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



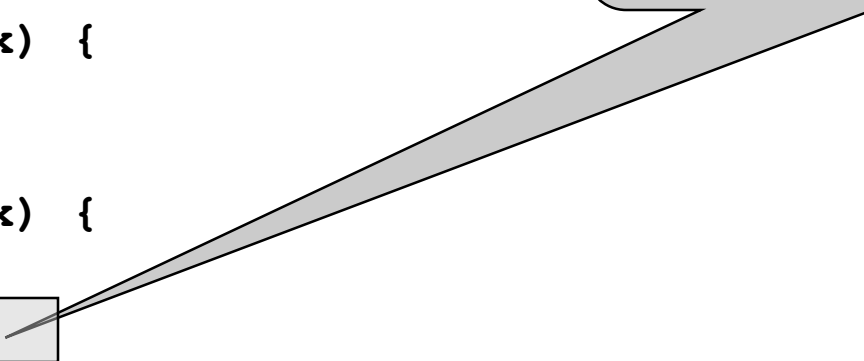
Execute the final block

Next statement;

# 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");
```

# 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

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.

```
1  public class InvalidRadiusException extends Exception {
2      private double radius;
3
4      /** Construct an exception */
5      public InvalidRadiusException(double radius) {
6          super("Invalid radius " + radius);
7          this.radius = radius;
8      }
9
10     /** Return the radius */
11     public double getRadius() {
12         return radius;
13     }
14 }
```

# Custom Exception Class Example

```
1 public class TestCircleWithCustomException {
2     public static void main(String[] args) {
3         try {
4             new CircleWithCustomException(5);
5             new CircleWithCustomException(-5);
6             new CircleWithCustomException(0);
7         }
8         catch (InvalidRadiusException ex) {
9             System.out.println(ex);
10        }
11
12        System.out.println("Number of objects created: " +
13            CircleWithCustomException.getNumberOfObjects());
14    }
15 }
16
17 class CircleWithCustomException {
18     /** The radius of the circle */
19     private double radius;
20
21     /** The number of objects created */
22     private static int numberOfObjects = 0;
23
24     /** Construct a circle with radius 1 */
25     public CircleWithCustomException() throws InvalidRadiusException {
26         this(1.0);
27     }
28
29     /** Construct a circle with a specified radius */
30     public CircleWithCustomException(double newRadius)
31         throws InvalidRadiusException {
32         setRadius(newRadius);
33         numberOfObjects++;
34     }
35 }
```

# Custom Exception Class Example

```
36  /** Return radius */
37  public double getRadius() {
38      return radius;
39  }
40
41  /** Set a new radius */
42  public void setRadius(double newRadius)
43      throws InvalidRadiusException {
44      if (newRadius >= 0)
45          radius = newRadius;
46      else
47          throw new InvalidRadiusException(newRadius);
48  }
49
50  /** Return numberOfObjects */
51  public static int getNumberOfObjects() {
52      return numberOfObjects;
53  }
54
55  /** Return the area of this circle */
56  public double findArea() {
57      return radius * radius * 3.14159;
58  }
59 }
```

```
InvalidRadiusException: Invalid radius -5.0
Number of objects created: 1
```

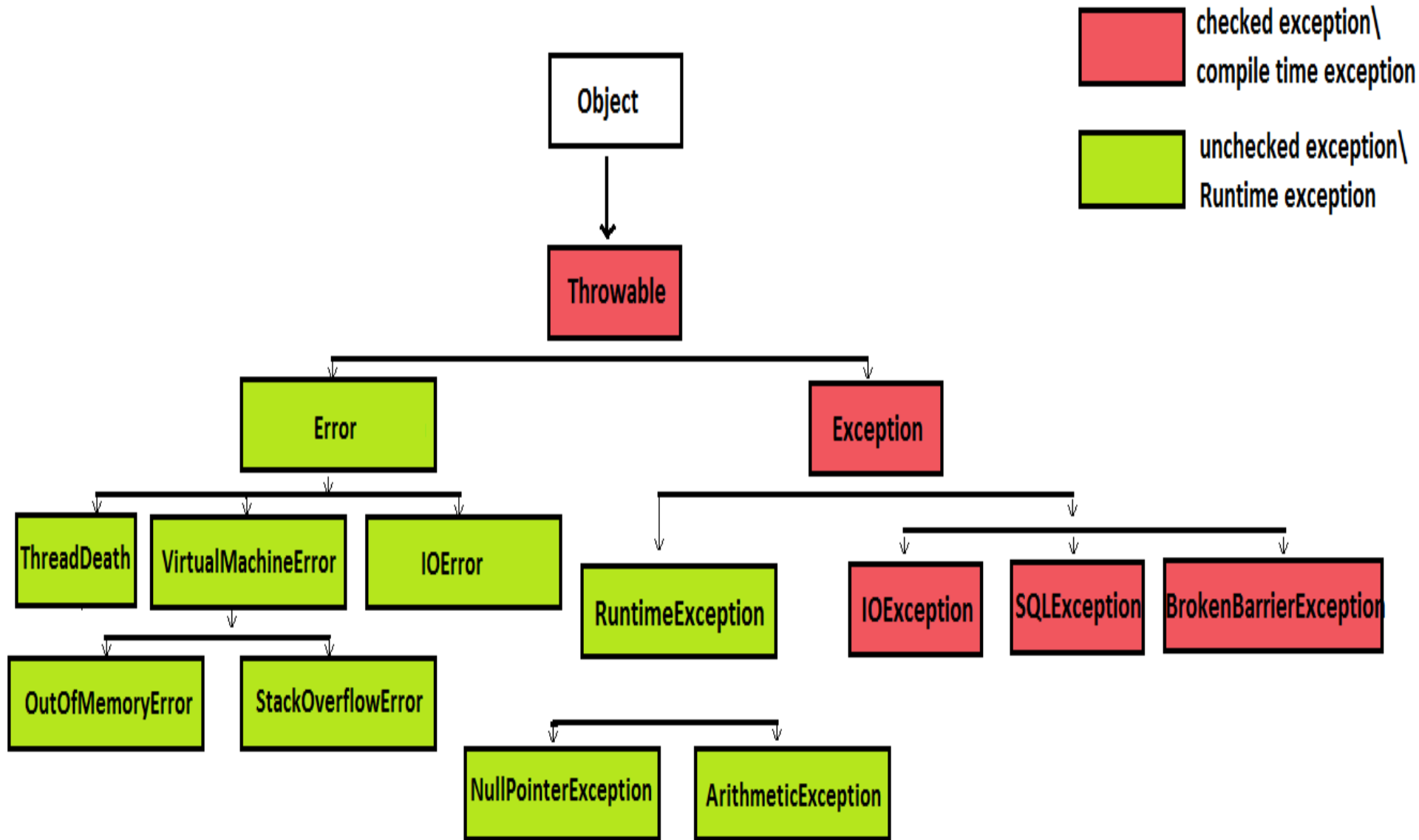


# How to create own exception


- ☞ **You can create your own exceptions in Java.**
- ☞ All **exceptions** must be a child of Throwable.
- ☞ **Checked exception:** If you want to **write** a checked **exception** that is automatically enforced by the Handle or Declare Rule, you need to extend the **Exception class**.
- ☞ **Unchecked exception:** If you want to **write** a runtime **exception**, you need to extend the RuntimeException **class**.
- ☞ Example: <https://www.baeldung.com/java-new-custom-exception>


# The Need for Custom Exceptions


- ☞ Java exceptions cover almost all general exceptions that are bound to happen in programming.
- ☞ However, we sometimes need to supplement these standard exceptions with our own.
- ☞ The main reasons for introducing custom exceptions are:
  - Business logic exceptions – Exceptions that are specific to the business logic and workflow. These help the application users or the developers understand what the exact problem is
  - To catch and provide specific treatment to a subset of existing Java exceptions
  - Java exceptions can be checked and unchecked. In the next sections, we'll cover both of these cases.




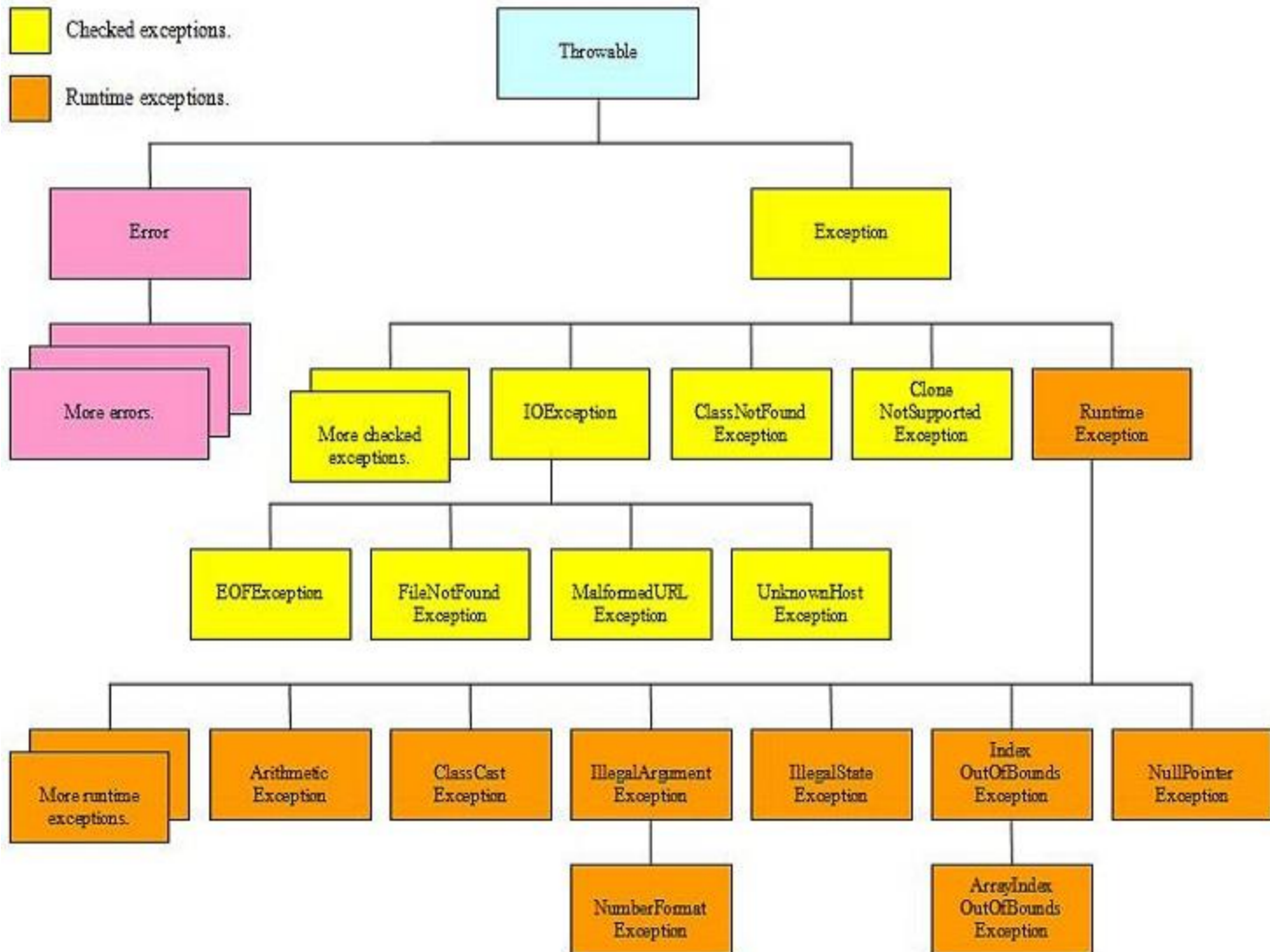
## Exception Hierarchy

 All exceptions inherit Throwable methods.

 Errors thrown by the JVM.

 Checked exceptions.

 Runtime exceptions.



# Blocks & Keywords used for exception handling

- ☞ **try:** The try block contains set of statements where an exception can occur.

```
try
{
    // statement(s) that might cause exception
}
```

- ☞ **catch :** Catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

```
catch
{
    // statement(s) that handle an exception
    // examples, closing a connection, closing
    // file, exiting the process after writing
    // details to a log file.
}
```

## Blocks & Keywords used for exception handling

- ➡ **throw:** Throw keyword is used to transfer control from try block to catch block.
- ➡ **throws:** Throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.
- ➡ **finally:** It is executed after catch block. We basically use it to put some common code when there are multiple catch blocks.

# Text Input / Output



# The File Class

The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The filename is a string. The File class is a wrapper class for the file name and its directory path.



# Obtaining file properties and manipulating file

java.io.File	
+File(pathname: String)	
+File(parent: String, child: String)	
+File(parent: File, child: String)	
+exists(): boolean	
+canRead(): boolean	
+canWrite(): boolean	
+isDirectory(): boolean	
+isFile(): boolean	
+isAbsolute(): boolean	
+isHidden(): boolean	
+getAbsolutePath(): String	
+getCanonicalPath(): String	
+getName(): String	
+getPath(): String	
+getParent(): String	
+lastModified(): long	
+length(): long	
+listFile(): File[]	
+delete(): boolean	
+renameTo(dest: File): boolean	
+mkdir(): boolean	
+mkdirs(): boolean	

Creates a **File** object for the specified path name. The path name may be a directory or a file.

Creates a **File** object for the child under the directory parent. The child may be a file name or a subdirectory.

Creates a **File** object for the child under the directory parent. The parent is a **File** object. In the preceding constructor, the parent is a string.

Returns true if the file or the directory represented by the **File** object exists.

Returns true if the file represented by the **File** object exists and can be read.

Returns true if the file represented by the **File** object exists and can be written.

Returns true if the **File** object represents a directory.

Returns true if the **File** object represents a file.

Returns true if the **File** object is created using an absolute path name.

Returns true if the file represented in the **File** object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.

Returns the complete absolute file or directory name represented by the **File** object.

Returns the same as **getAbsolutePath()** except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).

Returns the last name of the complete directory and file name represented by the **File** object. For example, new **File("c:\\book\\test.dat").getName()** returns **test.dat**.

Returns the complete directory and file name represented by the **File** object.

For example, new **File("c:\\book\\test.dat").getPath()** returns **c:\\book\\test.dat**.

Returns the complete parent directory of the current directory or the file represented by the **File** object. For example, new **File("c:\\book\\test.dat").getParent()** returns **c:\\book**.

Returns the time that the file was last modified.

Returns the size of the file, or 0 if it does not exist or if it is a directory.

Returns the files under the directory for a directory **File** object.

Deletes the file or directory represented by this **File** object. The method returns true if the deletion succeeds.

Renames the file or directory represented by this **File** object to the specified name represented in **dest**. The method returns true if the operation succeeds.

Creates a directory represented in this **File** object. Returns true if the the directory is created successfully.

Same as **mkdir()** except that it creates directory along with its parent directories if the parent directories do not exist.

# Problem: Explore File Properties

**Objective:** Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties.

```
1 public class TestFileClass {
2     public static void main(String[] args) {
3         java.io.File file = new java.io.File("image/us.gif");
4         System.out.println("Does it exist? " + file.exists());
5         System.out.println("The file has " + file.length() + " bytes");
6         System.out.println("Can it be read? " + file.canRead());
7         System.out.println("Can it be written? " + file.canWrite());
8         System.out.println("Is it a directory? " + file.isDirectory());
9         System.out.println("Is it a file? " + file.isFile());
10        System.out.println("Is it absolute? " + file.isAbsolute());
11        System.out.println("Is it hidden? " + file.isHidden());
12        System.out.println("Absolute path is " +
13            file.getAbsolutePath());
14        System.out.println("Last modified on " +
15            new java.util.Date(file.lastModified()));
16    }
17 }
```

# Objectives

- ➡ To discover file/directory properties, to delete and rename files/directories, and to create directories using the **File** class
- ➡ To write data to a file using the **PrintWriter** class
- ➡ To use try-with-resources to ensure that the resources are closed automatically
- ➡ To read data from a file using the **Scanner** class
- ➡ To understand how data is read using a **Scanner**
- ➡ To develop a program that replaces text in a file
- ➡ To read data from the Web

# Text I/O

A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. This section introduces how to read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

# Writing Data Using PrintWriter

## java.io.PrintWriter

+PrintWriter(filename: String)

Creates a PrintWriter for the specified file.

+print(s: String): void

Writes a string.

+print(c: char): void

Writes a character.

+print(cArray: char[]): void

Writes an array of character.

+print(i: int): void

Writes an int value.

+print(l: long): void

Writes a long value.

+print(f: float): void

Writes a float value.

+print(d: double): void

Writes a double value.

+print(b: boolean): void

Writes a boolean value.

Also contains the overloaded  
println methods.

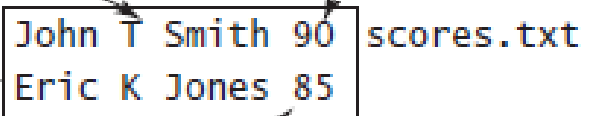
A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.

Also contains the overloaded  
printf methods.

The printf method was introduced in §3.6, “Formatting Console Output and Strings.”

# Writing Data Using PrintWriter

```
1 public class WriteData {
2     public static void main(String[] args) throws IOException {
3         java.io.File file = new java.io.File("scores.txt");
4         if (file.exists()) {
5             System.out.println("File already exists");
6             System.exit(1);
7         }
8
9         // Create a file
10        java.io.PrintWriter output = new java.io.PrintWriter(file);
11
12        // Write formatted output to the file
13        output.print("John T Smith ");
14        output.println(90);
15        output.print("Eric K Jones ");
16        output.println(85);
17
18        // Close the file
19        output.close();
20    }
21 }
```



John T Smith 90  
Eric K Jones 85

scores.txt

# Try-with-resources

Programmers often forget to close the file. JDK 7 provides the following new try-with-resources syntax that automatically closes the files.

**try** (declare and create resources) {

    Use the resource to process the file;

}

```
1  public class WriteDataWithAutoClose {
2      public static void main(String[] args) throws Exception {
3          java.io.File file = new java.io.File("scores.txt");
4          if (file.exists()) {
5              System.out.println("File already exists");
6              System.exit(0);
7          }
8
9          try (
10             // Create a file
11             java.io.PrintWriter output = new java.io.PrintWriter(file);
12         ) {
13             // Write formatted output to the file
14             output.print("John T Smith ");
15             output.println(90);
16             output.print("Eric K Jones ");
17             output.println(85);
18         }
19     }
20 }
```

# Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.



# Reading Data Using Scanner

```
1 import java.util.Scanner;
2
3 public class ReadData {
4     public static void main(String[] args) throws Exception {
5         // Create a File instance
6         java.io.File file = new java.io.File("scores.txt");
7
8         // Create a Scanner for the file
9         Scanner input = new Scanner(file);
10
11        // Read data from a file
12        while (input.hasNext()) {
13            String firstName = input.next();
14            String mi = input.next();
15            String lastName = input.next();
16            int score = input.nextInt();
17            System.out.println(
18                firstName + " " + mi + " " + lastName + " " + score);
19        }
20
21        // Close the file
22        input.close();
23    }
24 }
```

scores.txt

John	T	Smith	90
Eric	K	Jones	85

# Problem: Replacing Text

Write a class named ReplaceText that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

# Problem: Replacing Text

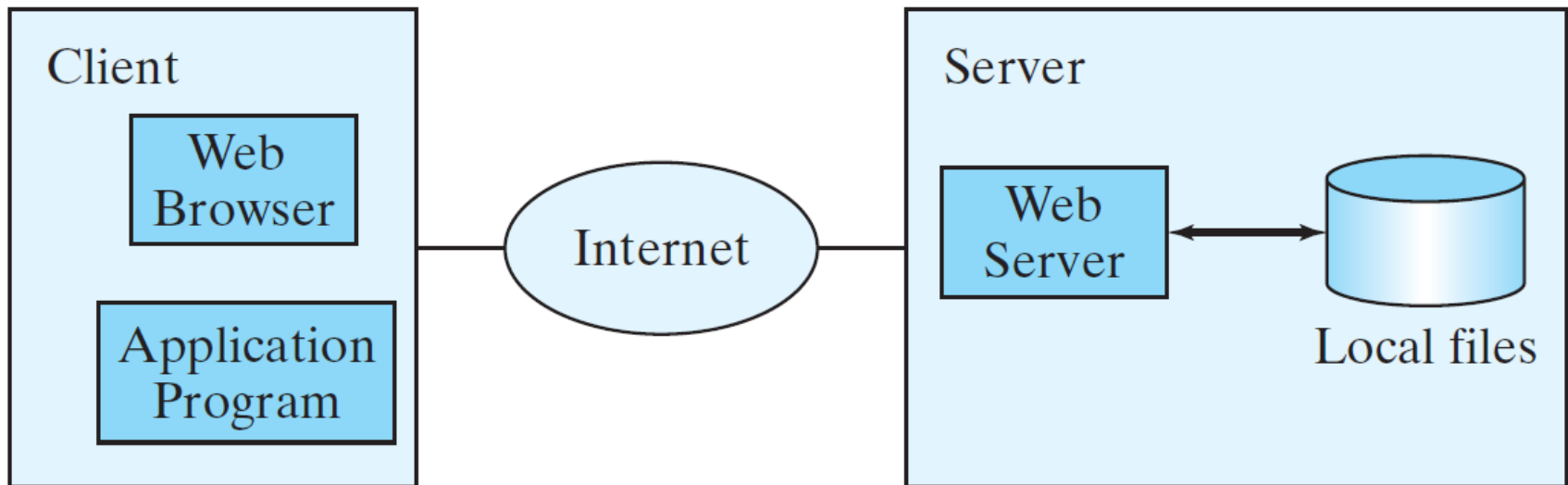
```
1  import java.io.*;
2  import java.util.*;
3
4  public class ReplaceText {
5      public static void main(String[] args) throws Exception {
6          // Check command line parameter usage
7          if (args.length != 4) {
8              System.out.println(
9                  "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
10             System.exit(1);
11         }
12
13         // Check if source file exists
14         File sourceFile = new File(args[0]);
15         if (!sourceFile.exists()) {
16             System.out.println("Source file " + args[0] + " does not exist");
17             System.exit(2);
18         }
19
20         // Check if target file exists
21         File targetFile = new File(args[1]);
22         if (targetFile.exists()) {
23             System.out.println("Target file " + args[1] + " already exists");
24             System.exit(3);
25         }
26     }
27 }
```

# Problem: Replacing Text

```
26
27     try (
28         // Create input and output files
29         Scanner input = new Scanner(sourceFile);
30         PrintWriter output = new PrintWriter(targetFile);
31     ) {
32         while (input.hasNext()) {
33             String s1 = input.nextLine();
34             String s2 = s1.replaceAll(args[2], args[3]);
35             output.println(s2);
36         }
37     }
38 }
39 }
```

# Reading Data from the Web

Just like you can read data from a file on your computer, you can read data from a file on the Web.



# Reading Data from the Web

```
URL url = new URL("www.google.com/index.html");
```

After a **URL** object is created, you can use the **openStream()** method defined in the **URL** class to open an input stream and use this stream to create a **Scanner** object as follows:

```
Scanner input = new Scanner(url.openStream());
```

# Reading Data from the Web

```
1 import java.util.Scanner;
2
3 public class ReadFileFromURL {
4     public static void main(String[] args) {
5         System.out.print("Enter a URL: ");
6         String urlString = new Scanner(System.in).next();
7
8         try {
9             java.net.URL url = new java.net.URL(urlString);
10            int count = 0;
11            Scanner input = new Scanner(url.openStream());
12            while (input.hasNext()) {
13                String line = input.nextLine();
14                count += line.length();
15            }
16
17            System.out.println("The file size is " + count + " characters");
18        }
19        catch (java.net.MalformedURLException ex) {
20            System.out.println("Invalid URL");
21        }
22        catch (java.io.IOException ex) {
23            System.out.println("I/O Errors: no such file");
24        }
25    }
26 }
```

Enter a URL: <http://cs.armstrong.edu/liang/data/Lincoln.txt>  
The file size is 1469 characters

Enter