Lecture 4: Exception Handling and Text IO (Ch 14)

Adapted by Fangzhen Lin for COMP3021 from Y. Danial Liang's PowerPoints for Introduction to Java Programming, Comprehensive Version, 9/E, Pearson, 2013.

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 lecture.

Objectives

- To get an overview of exceptions and exception handling (§14.2).
- To explore the advantages of using exception handling (§14.2).
- To distinguish exception types: **Error** (fatal) vs. **Exception** (nonfatal) and checked vs. unchecked (§14.3).
- To declare exceptions in a method header (§14.4.1).
- To throw exceptions in a method (§14.4.2).
- To write a **try-catch** block to handle exceptions (§14.4.3).
- To explain how an exception is propagated (§14.4.3).
- To obtain information from an exception object (§14.4.4).
- To develop applications with exception handling (§14.4.5).
- To use the **finally** clause in a **try-catch** block (§14.5).
- To use exceptions only for unexpected errors (§14.6).
- To rethrow exceptions in a **catch** block (§14.7).
- To create chained exceptions (§14.8).
- To define custom exception classes (§14.9).
- To discover file/directory properties, to delete and rename files/directories, and to create directories using the **File** class (§14.10).
- To write data to a file using the **PrintWriter** class (§14.11.1).
- To read data from a file using the **Scanner** class (§14.11.2).
- To understand how data is read using a **Scanner** (§14.11.3).
- To develop a program that replaces text in a file (§14.11.4).
- To open files using a file dialog box (§14.12).
- To read data from the Web (§14.13).

Exception-Handling Overview

Show runtime error

Quotient

Fix it using an if statement

QuotientWithIf

With a method

QuotientWithMethod

Exception Advantages

QuotientWithException

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:

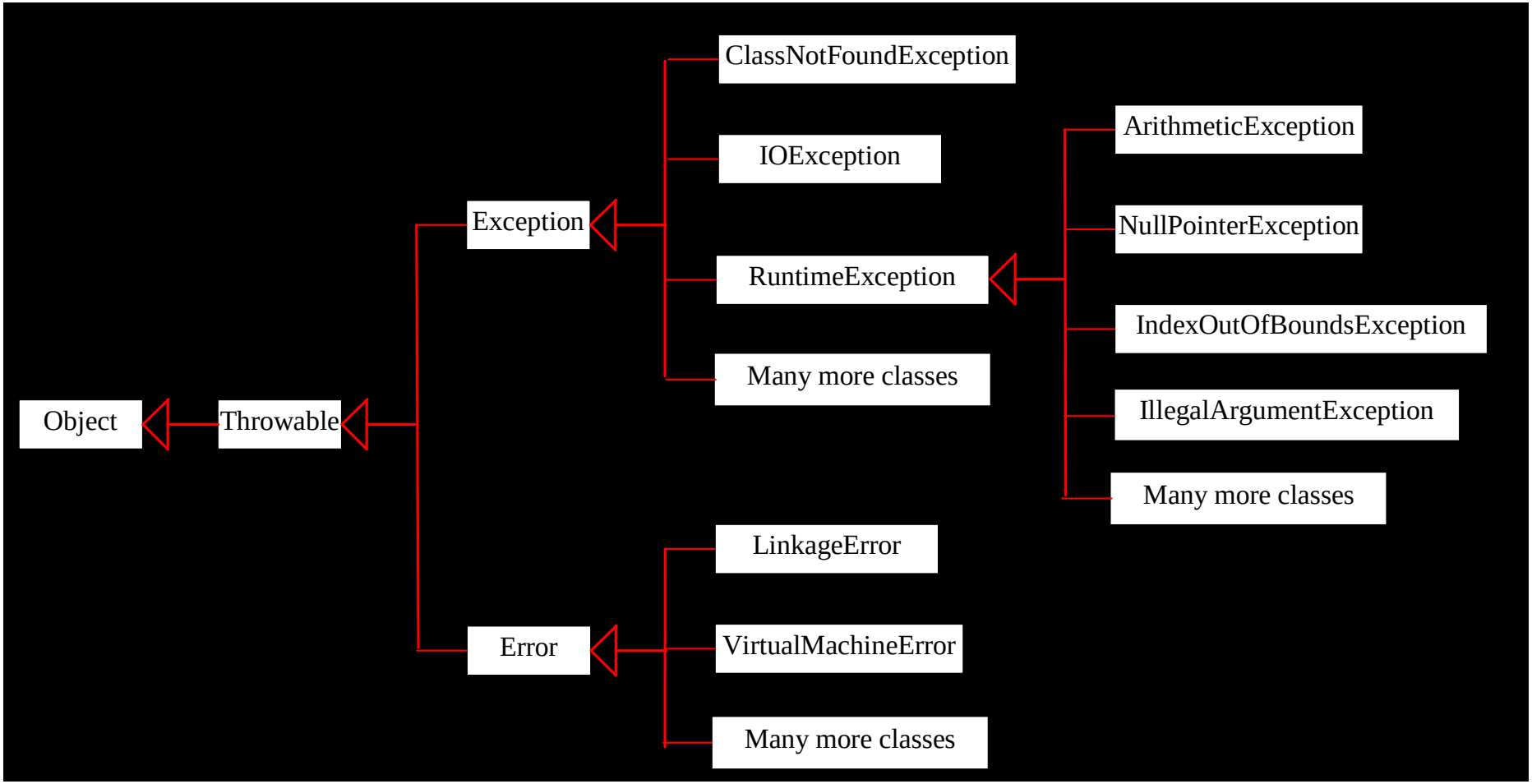
- A callee can detect an error but does not know what to do with it;
- A caller needs the callee to tell it whether any error has occured and decides what to do based on the type of errors.

Handling InputMismatchException

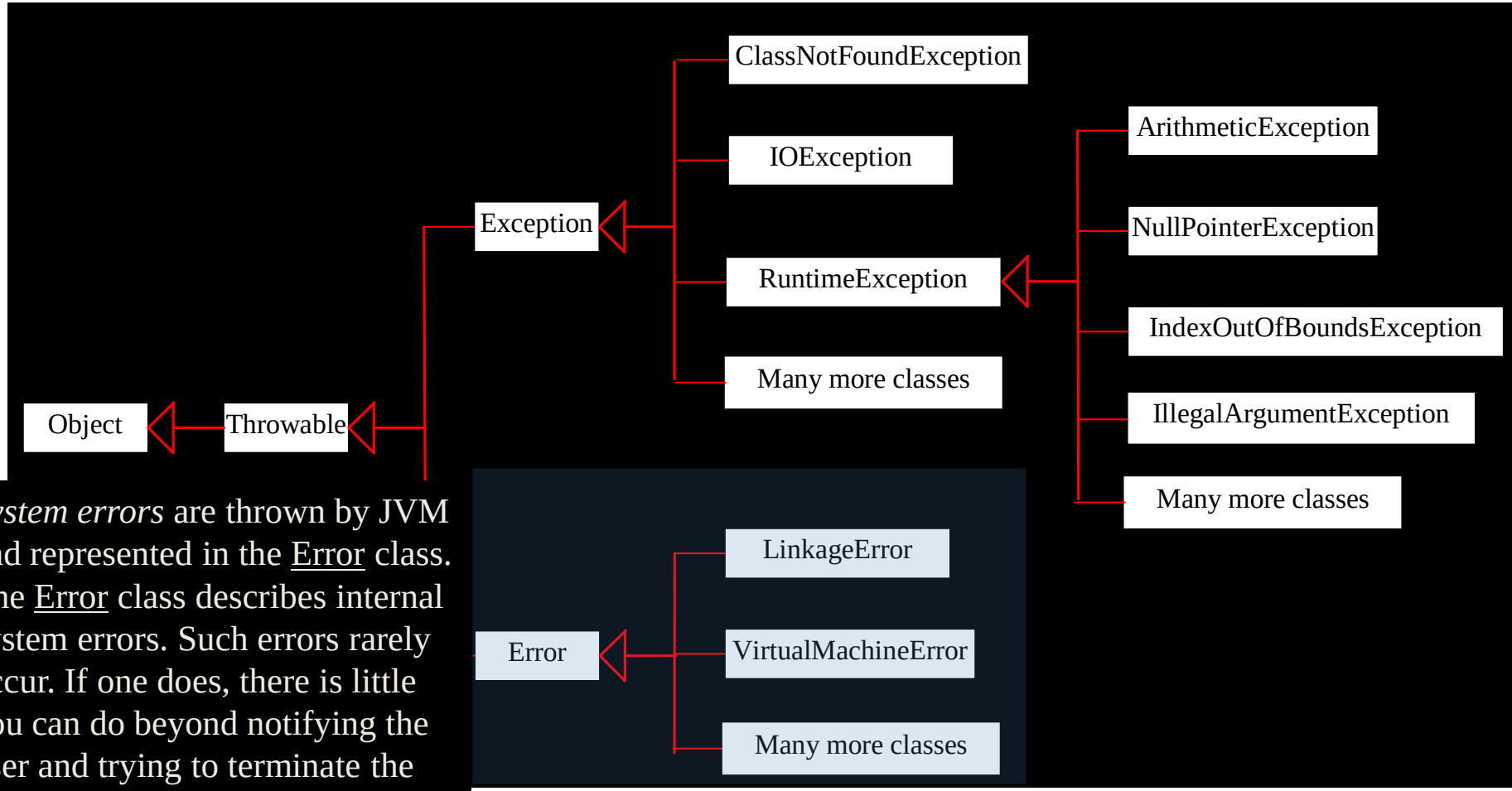
InputMismatchExceptionDemo

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

Exception Types



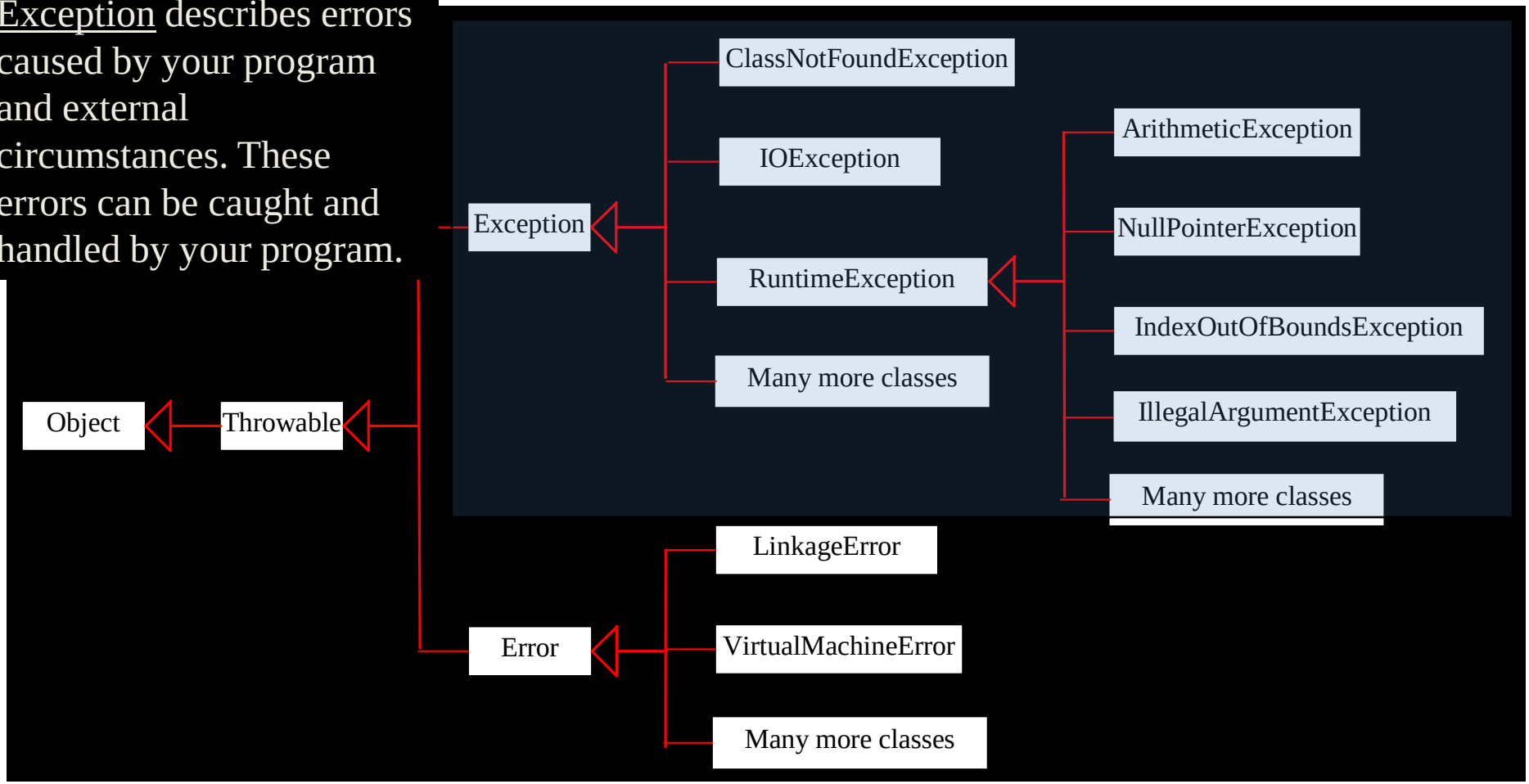
System Errors



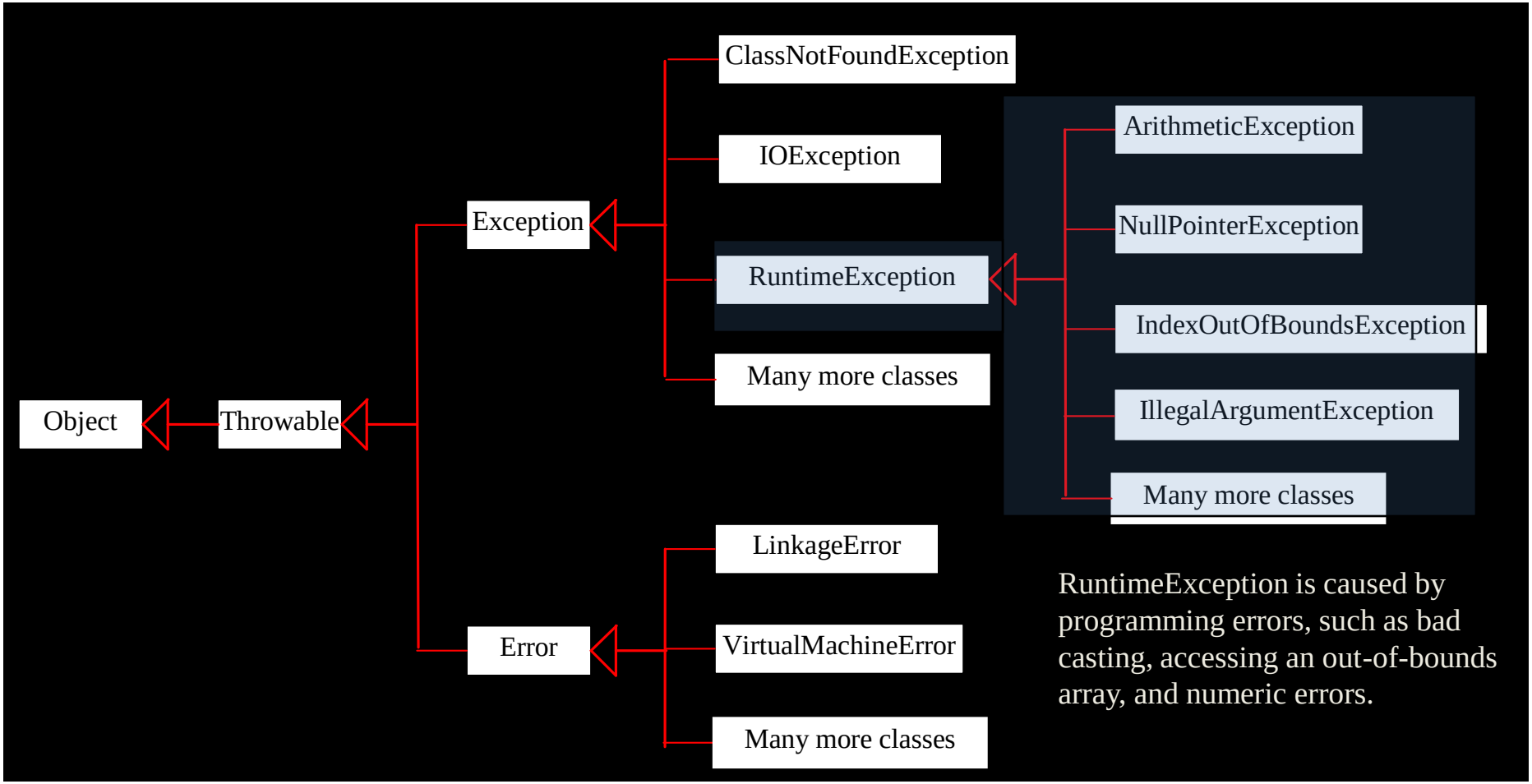
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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



Runtime Exceptions



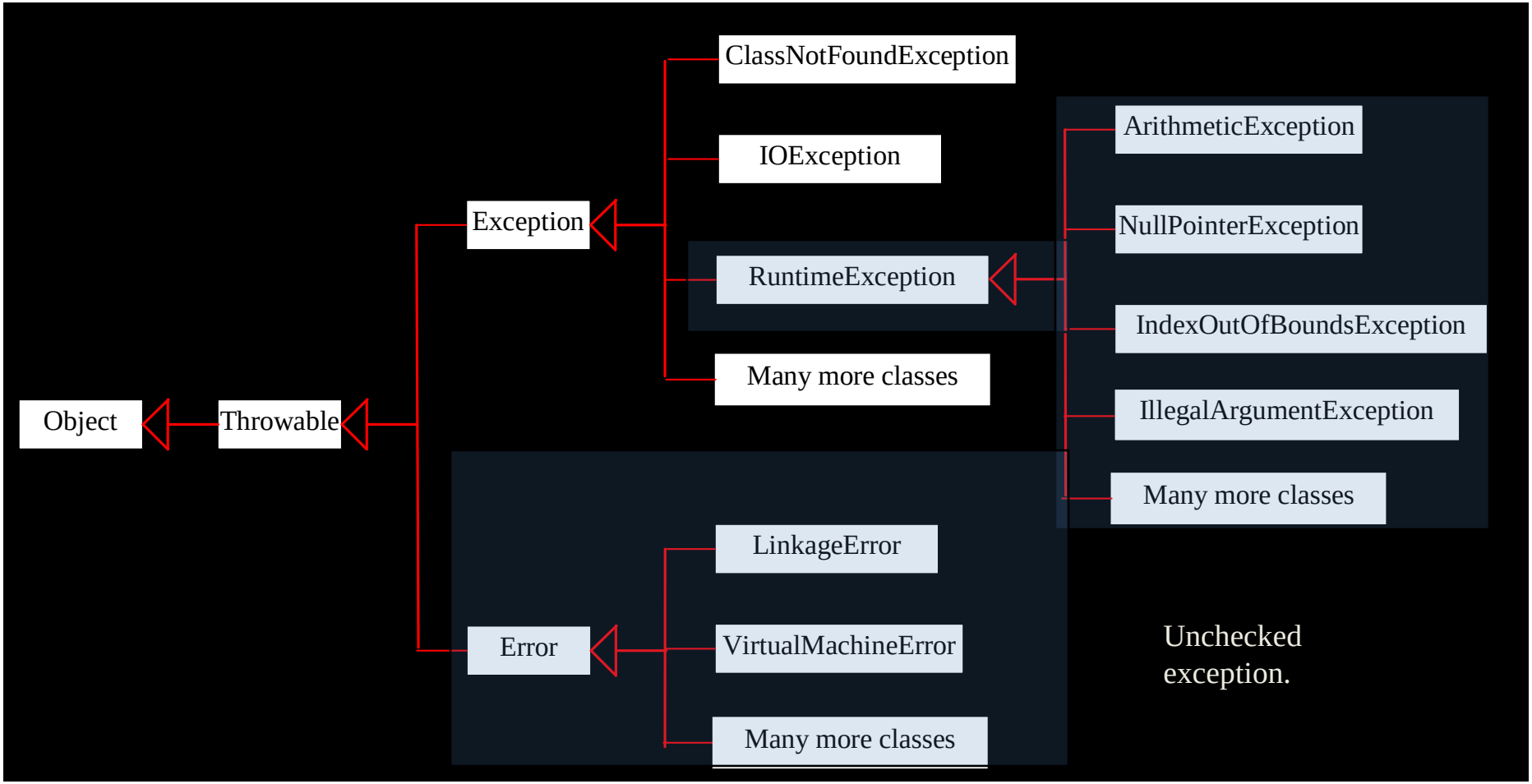
Checked Exceptions vs. Unchecked Exceptions

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.

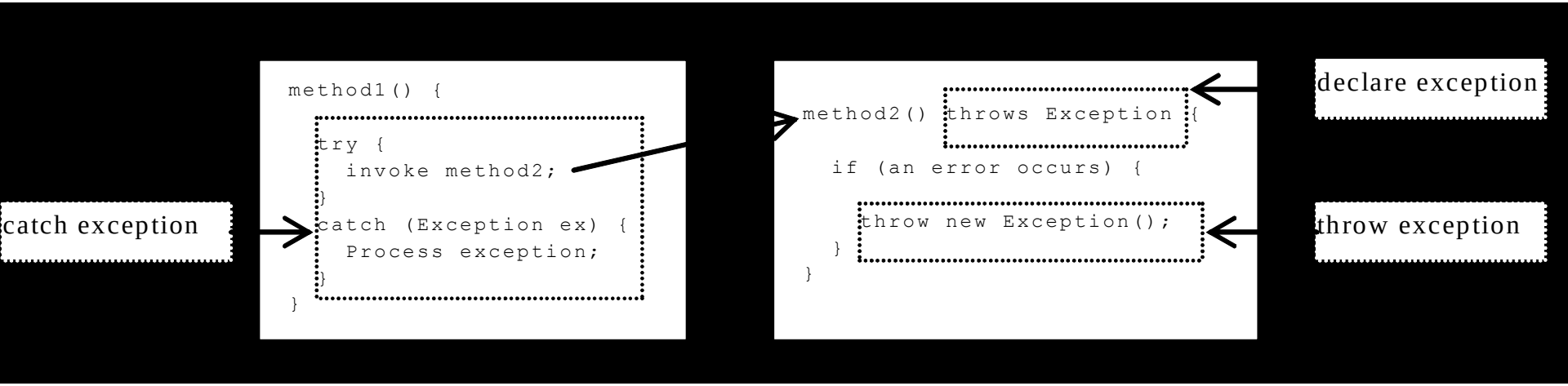
Unchecked Exceptions

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;
```

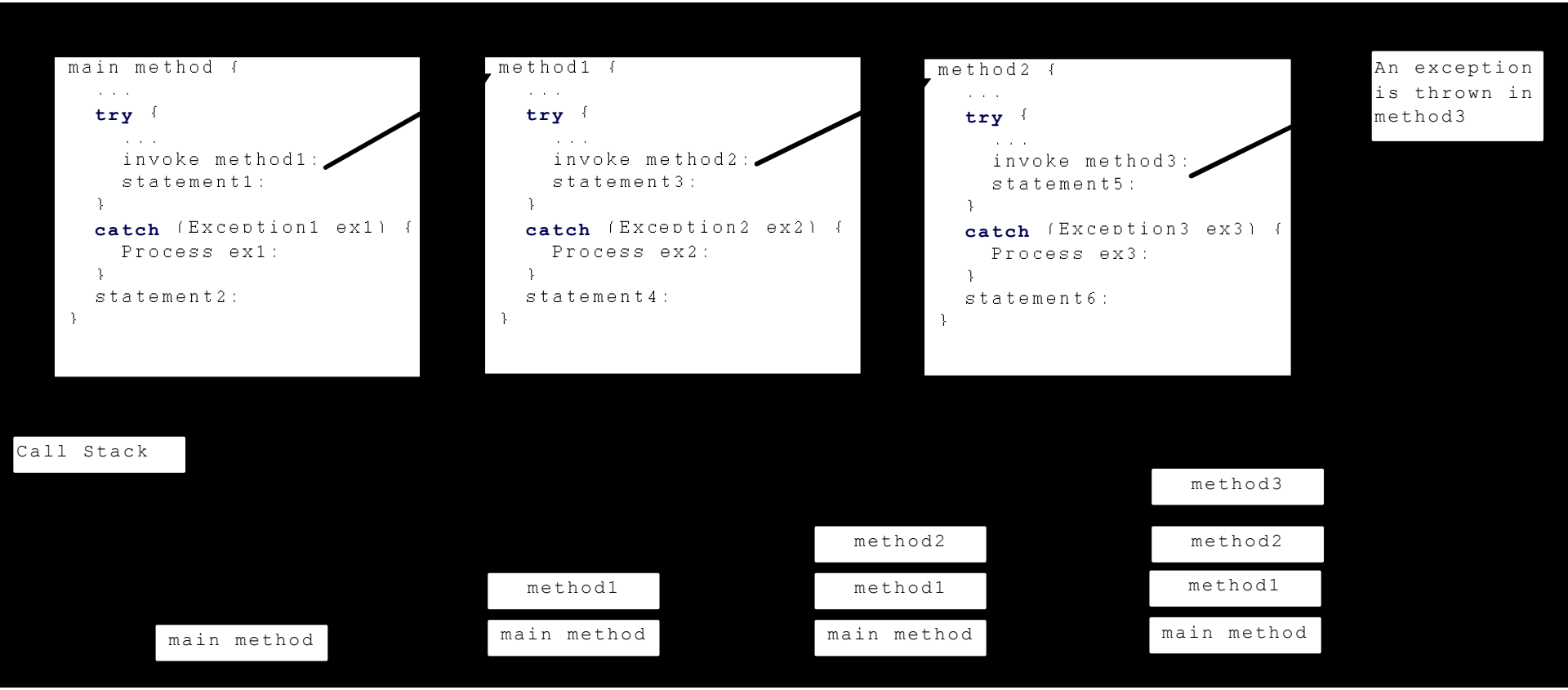
Throwing Exceptions Example

```
   /** 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



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.

TestCircleWithException

CircleWithException

The `finally` Clause

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

animation

Trace a Program Execution

Suppose no exceptions in the statements

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

Next statement;
```

animation

Trace a Program Execution

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

Next statement;
```

The final block is always executed

animation

Trace a Program Execution

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

Next statement;
```

Next statement in the method is executed

animation

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

animation

Trace a Program Execution

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

Next statement;
```

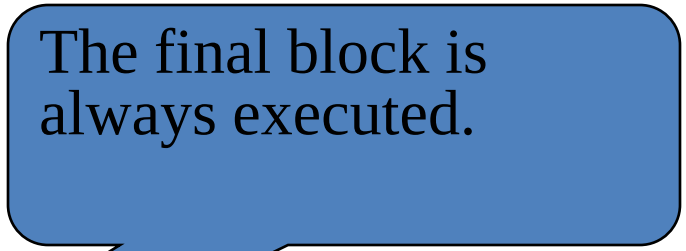
The exception is handled.

animation

Trace a Program Execution

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

Next statement;
```



animation

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.

animation

Trace a Program Execution

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

Next statement;
```

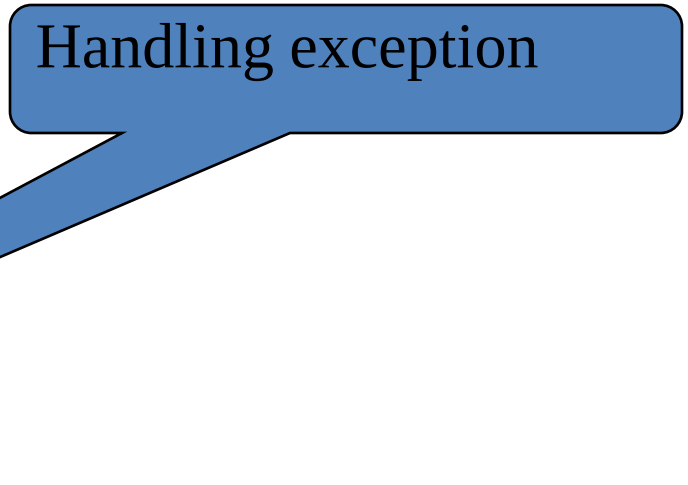
statement2 throws an exception of type Exception2.

animation

Trace a Program Execution

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

Next statement;
```

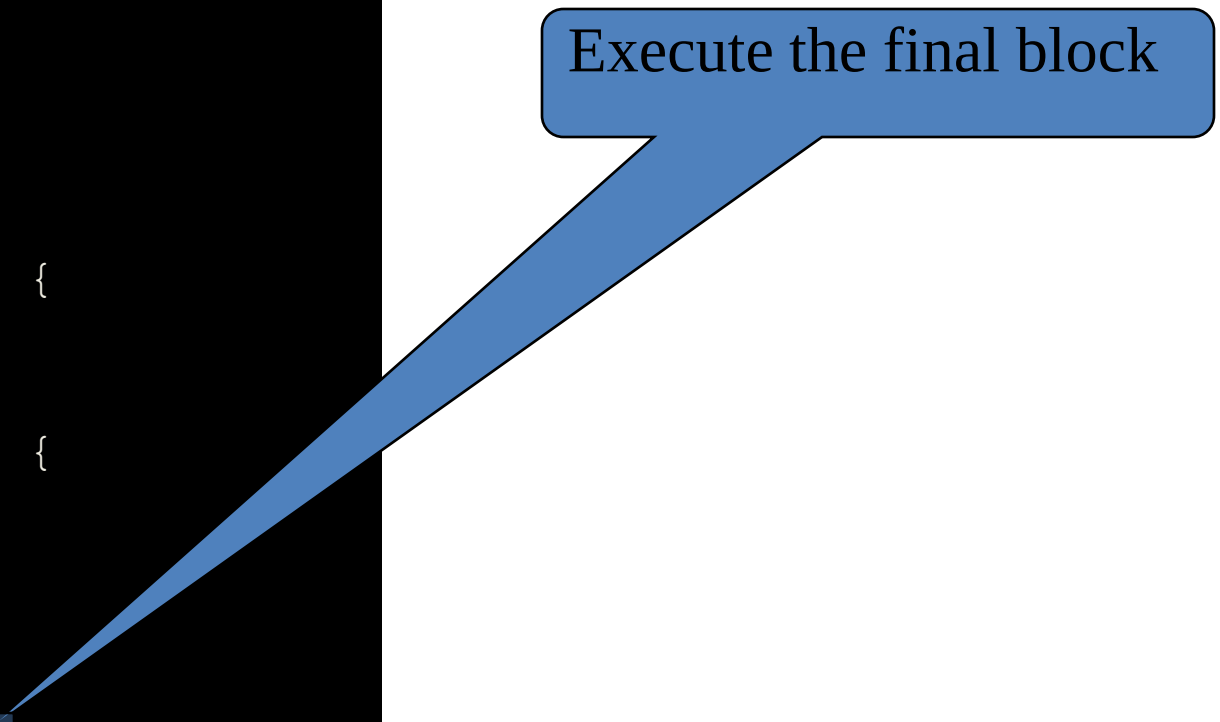


animation

Trace a Program Execution

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

Next statement;
```



animation

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

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

CircleWithRadiusException

TestCircleWithRadiusException

The File Class

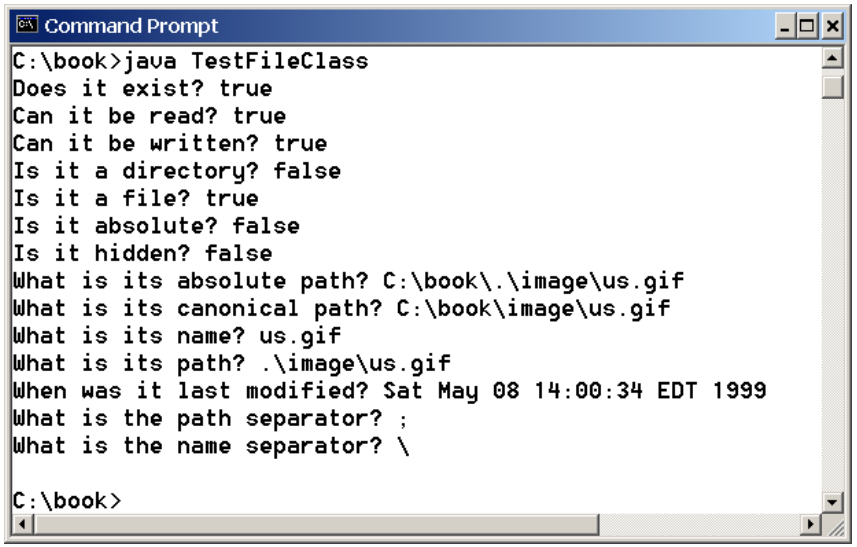
The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The filename is a string. The File class is a wrapper class for the file name and its directory path.

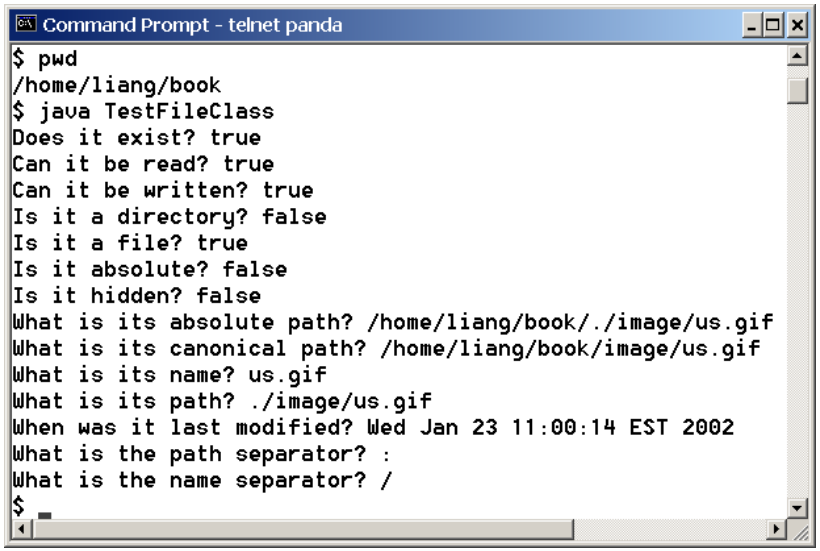
Obtaining file properties and manipulating file

java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified pathname. The pathname may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a filename or a subdirectory.
`+File(parent: File, child: String)`	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.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	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.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the pathname, resolves symbolic links (on UNIX), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	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`.
`+getPath(): String`	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`.
`+getParent(): String`	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`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFiles(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object. The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	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.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with it 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. The following figures show a sample run of the program on Windows and on Unix.





TestFileClass

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."

WriteData

Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned 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.

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

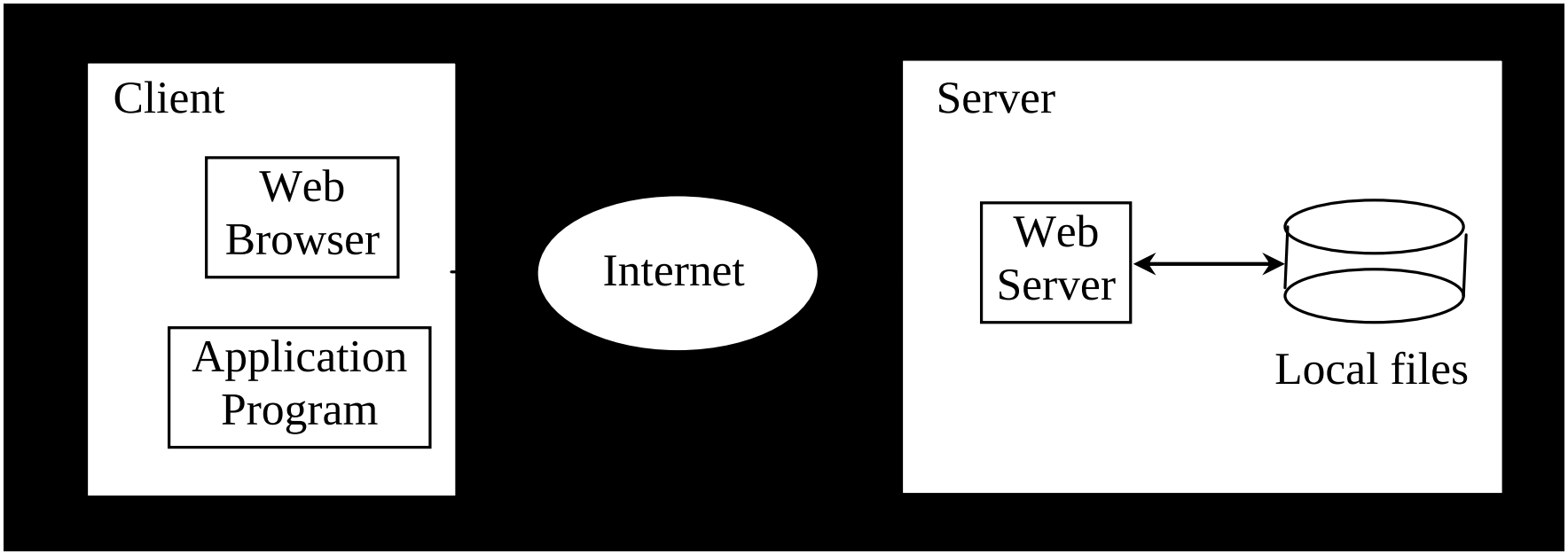
```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

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

ReplaceText

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());

ReadFileFromURL

Case Study: Web Crawler

This case study develops a program that travels the Web by following hyperlinks.

Case Study: Web Crawler

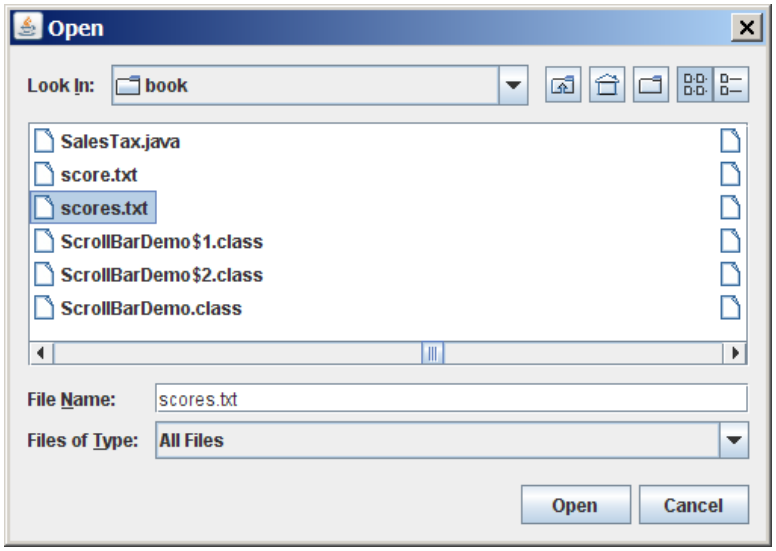
The program follows the URLs to traverse the Web. To make sure that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing and the other stores the URLs that have already been traversed. The algorithm for this program can be described as follows:

Case Study: Web Crawler

```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty {
      Remove a URL from listOfPendingURLs;
      if this URL is not in listOfTraversedURLs {
        Add it to listOfTraversedURLs;
        Display this URL;
        Exit the while loop when the size of S is equal to 100.
        Read the page from this URL and for each URL contained in the page {
          Add it to listOfPendingURLs if it is not is listOfTraversedURLs;
        }
      }
    }
```

WebCrawler

(GUI) File Dialogs



ReadFileUsingJFileChooser