UNIVERSITÉ Concordia UNIVERSITY

CENTRE FOR CONTINUING EDUCATION

DESKTOP APPLICATION DEVELOPMENT WITH JAVA – CEJV569

Lecture #3

I/O

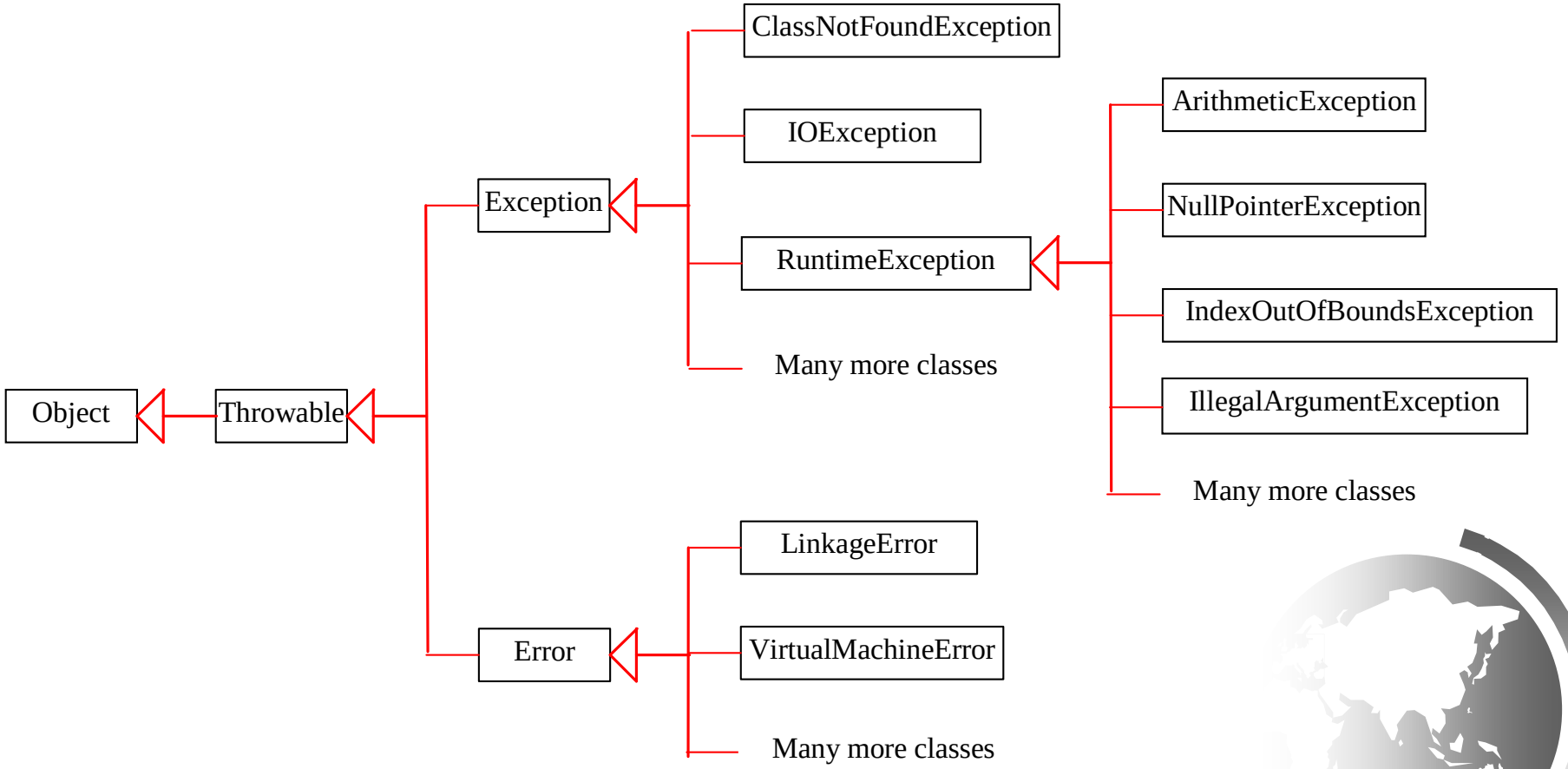
Reading from Web

Motivations

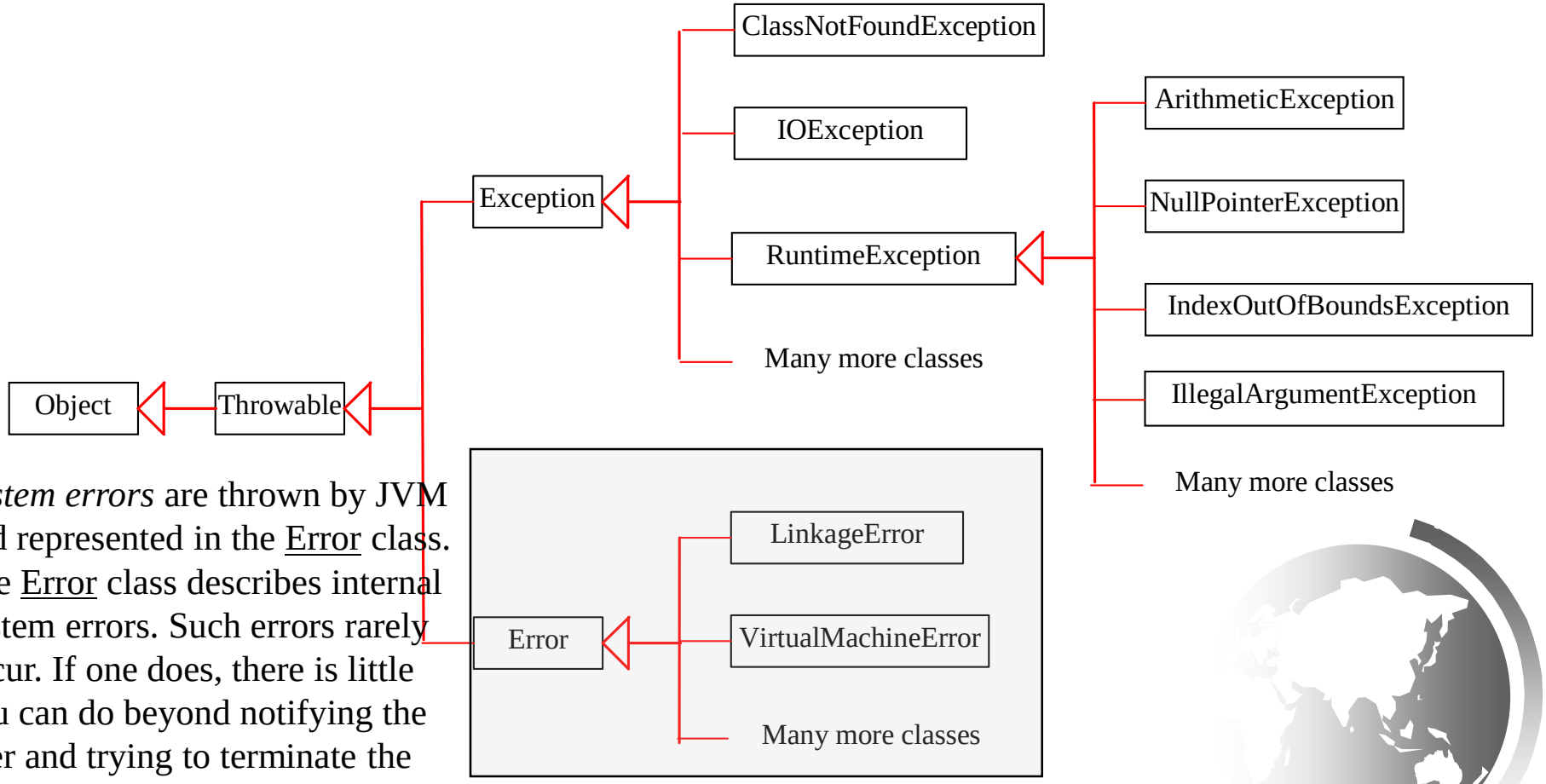
When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully?

The solution is ***handling Exception***.

Exception Types



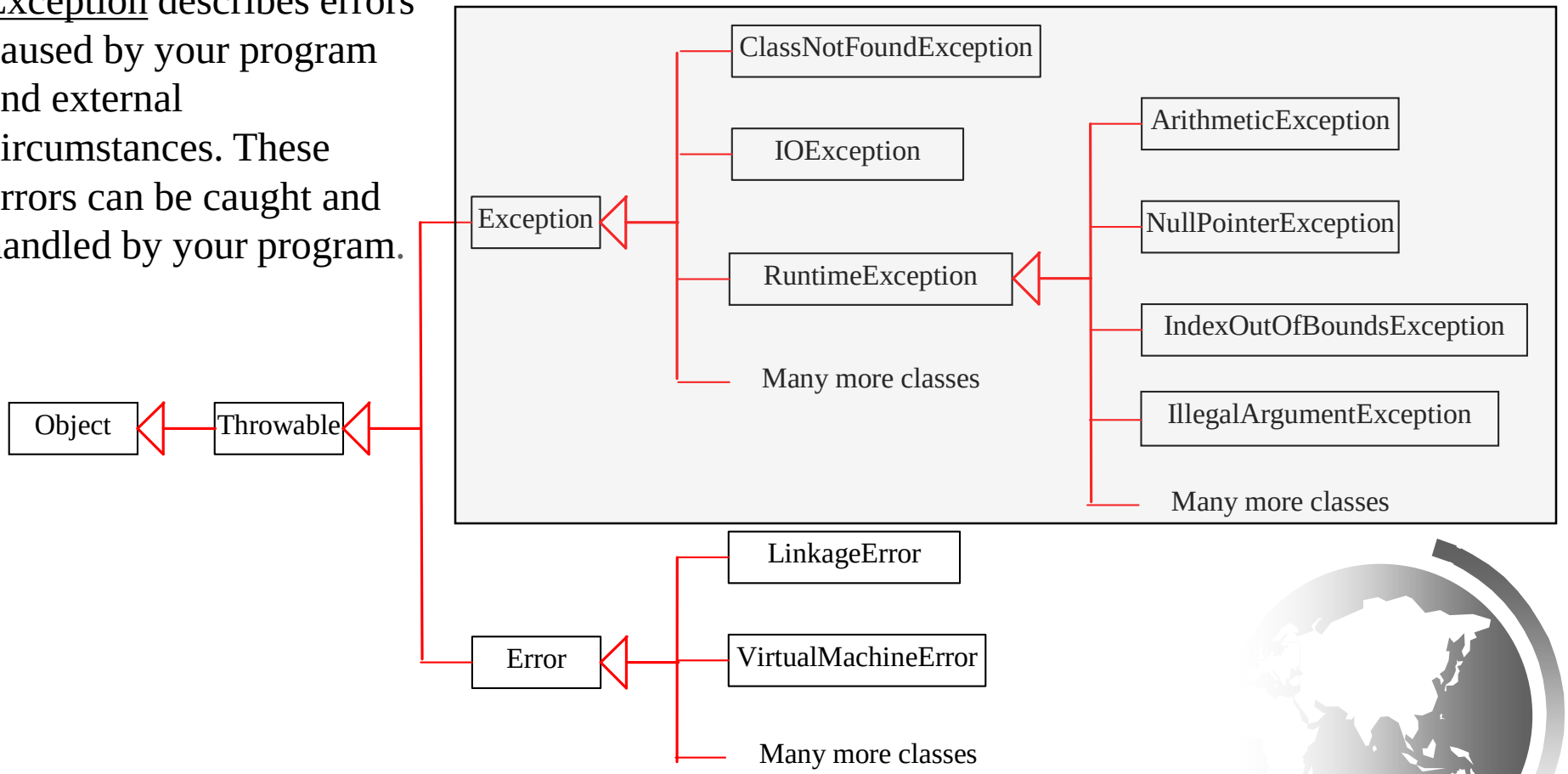
System Errors



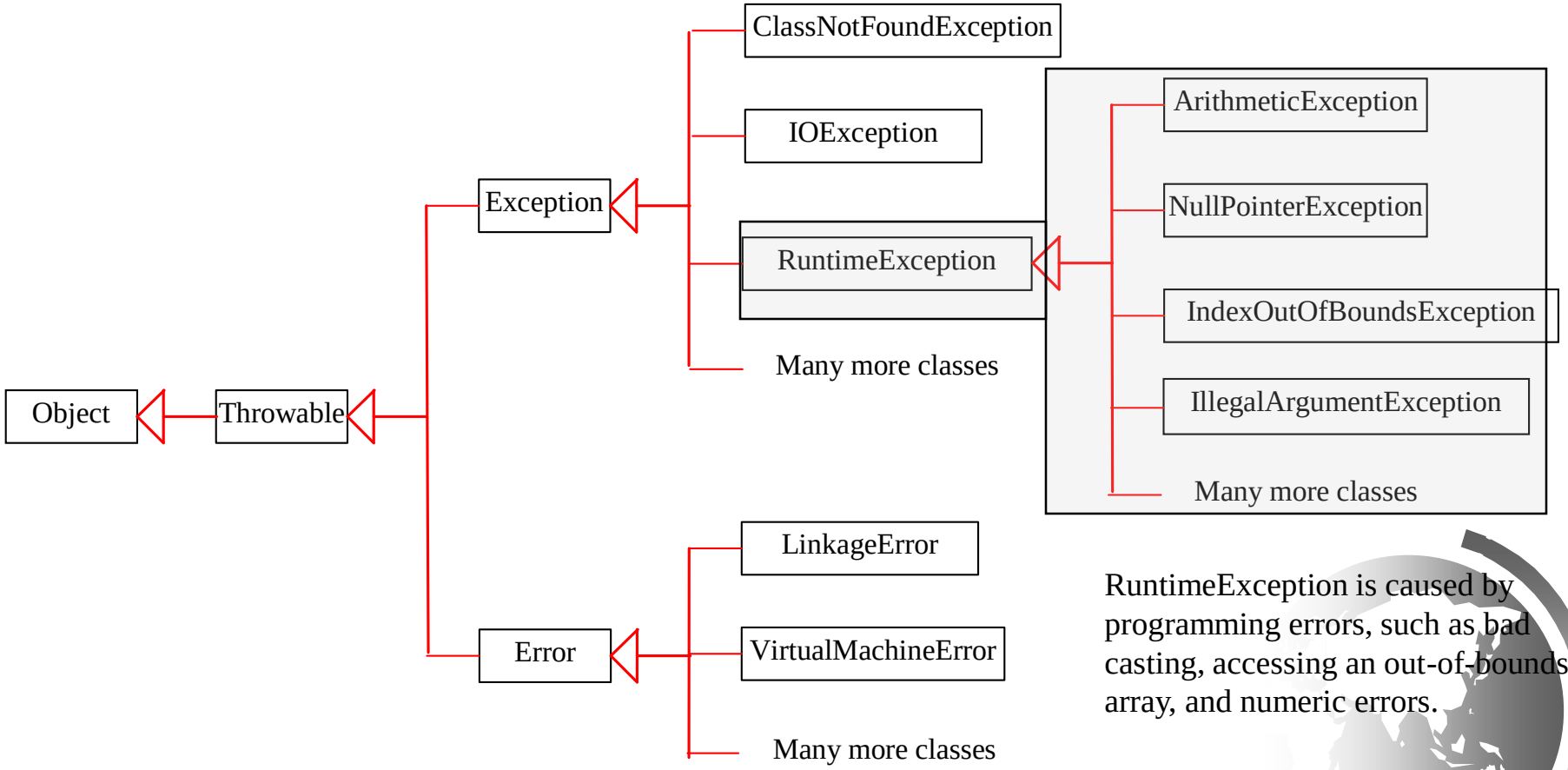
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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



Runtime Exceptions



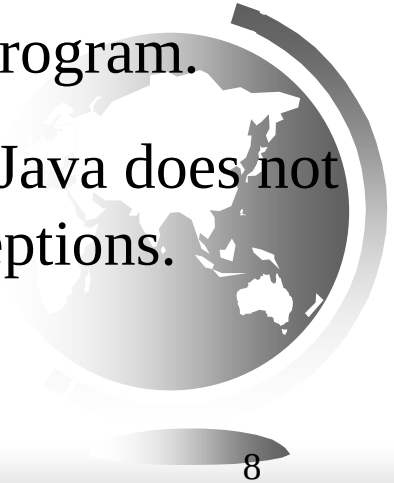
RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Checked Exceptions vs. Unchecked Exceptions

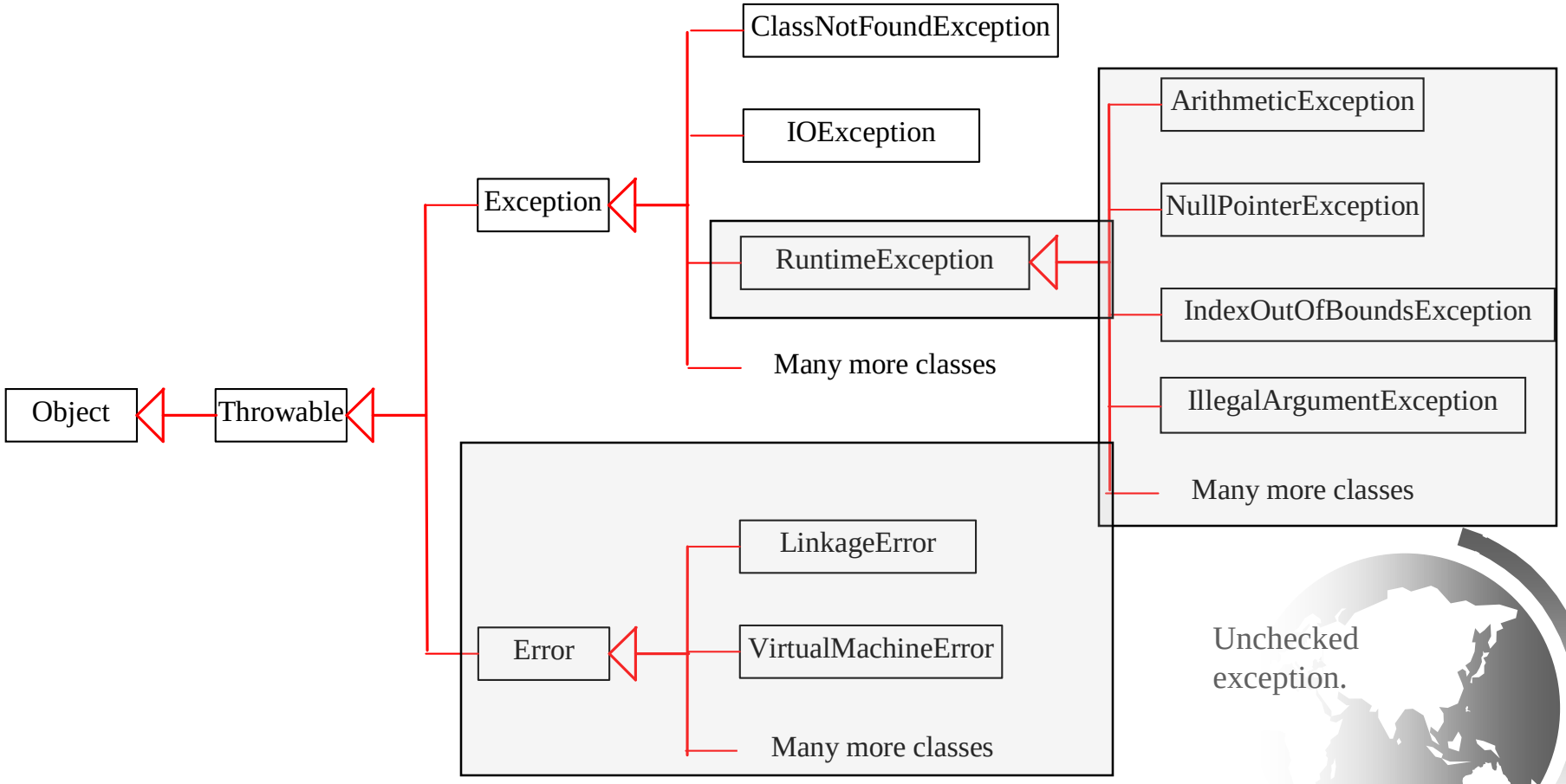
- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

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



Unchecked 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;
```

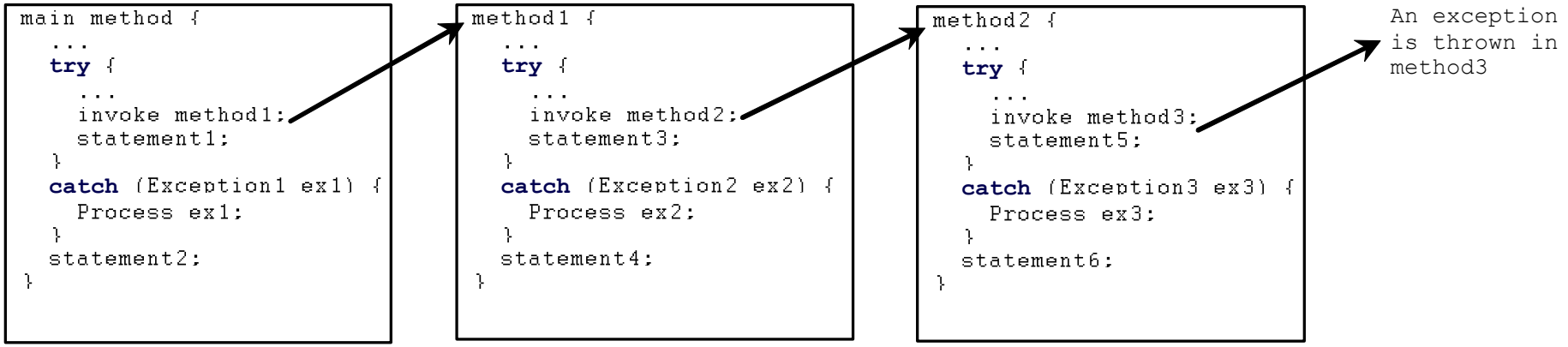
Throwing Exceptions Example

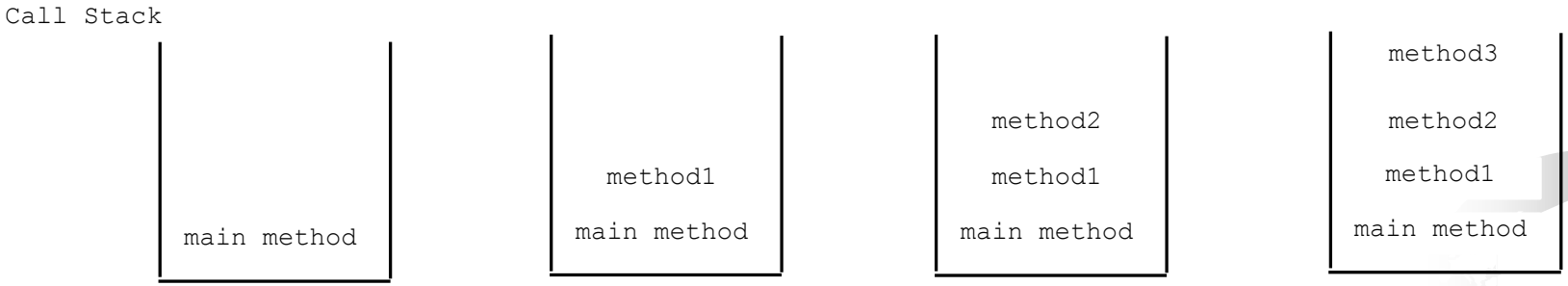
```
   /** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

Catching Exceptions

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

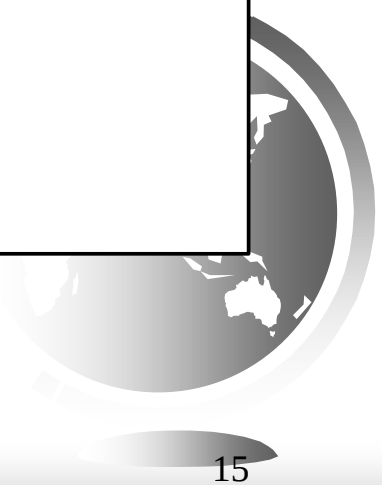
(a)

```
void p1() throws IOException {

  p2();

}
```

(b)



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;
```

The final block is always executed.

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;
```

Handling exception

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;
```

Execute the final block

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

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.

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."

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.

Exercise 13: Replacing Text

Write a class named ReplaceText that replaces a string in a text file with a new string.

For example, your program replaces all the occurrences of public by protected in ReadData.java and saves the new file in t.txt.

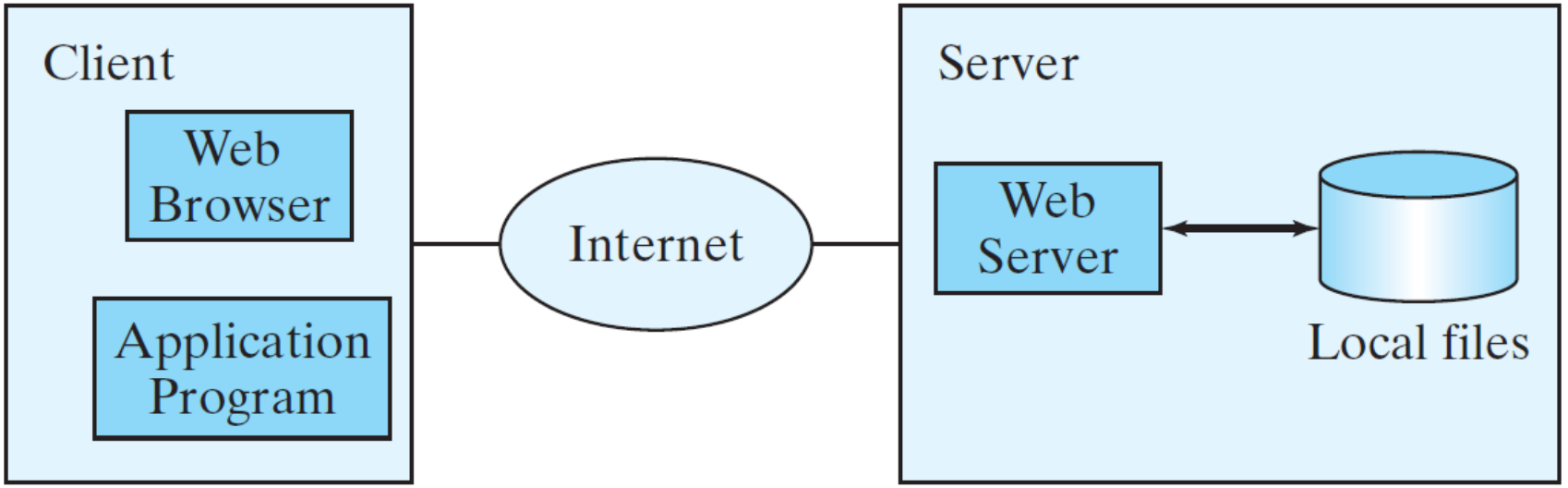
Exercise 14

Remove Text

- Write a complete program that removes all the occurrences of a specified string from a text file. For example, invoking Exercise14.java removes the string “John” from the specified file. Your program should get the specified world and the address of file from the users.

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

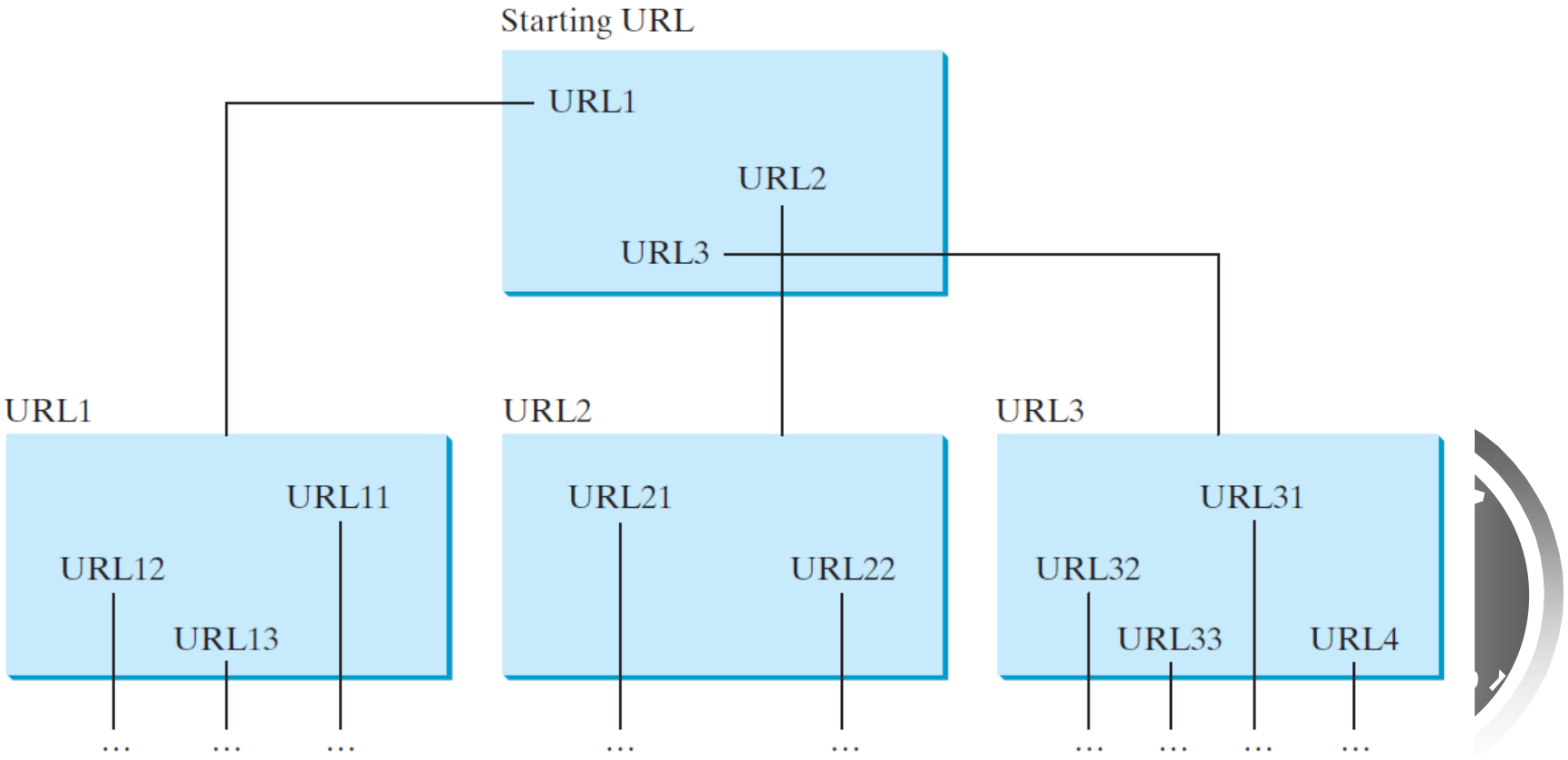
URL url = **new** URL("**www.google.com/index.html**");

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());

Case Study: Web Crawler

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



Case Study: Web Crawler

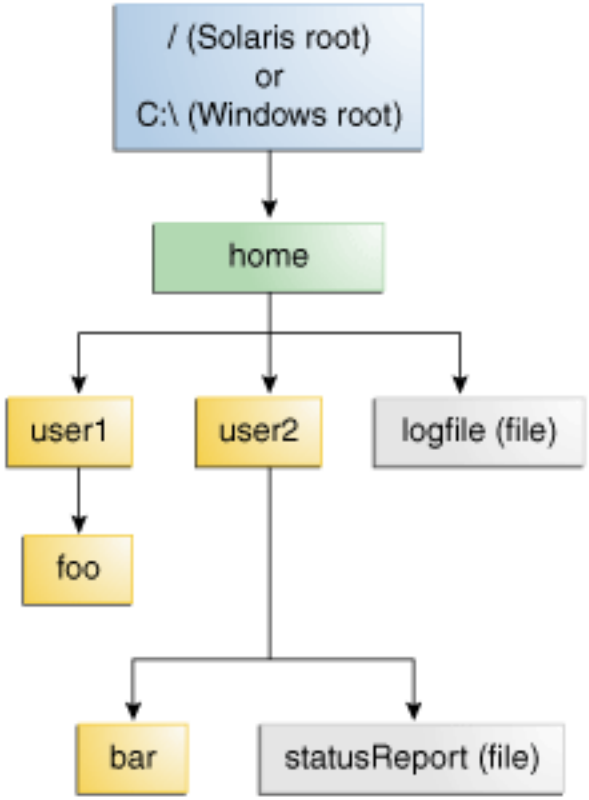
The program follows the URLs to traverse the Web. To avoid that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing and the other stores the URLs that have already been traversed. The algorithm for this program can be described as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

What Is a Path?

- A file is identified by its path through the file system, beginning from the root node



Relative or Absolute?

- A path is either relative or absolute.
- An absolute path always contains the root element and the complete directory list required to locate the file.
- C:\home\sally\statusReport.doc
- A relative path needs to be combined with another path in order to access a file.
- joe\foo.txt
- Without more information, a program cannot reliably locate the joe\foo.txt directory in the file system.

The Path Class

- Programmatic representation of a path in the file system
- Contains the file name and directory list used to construct the path
- Used to examine, locate, and manipulate files.
- Not system independent
- Cannot compare a Path from a Solaris file system and expect it to match a Path from a Windows file system
- File or directory corresponding to the Path might not exist

Creating a Path

- Use one of the following get methods from the Paths (note the plural) helper class:

```
Path p1 = Paths.get(“C:\tmp\foo");
Path p2 = Paths.get(var);
Path p3 =

Paths.get(URI.create("file:///C:/joe/FileTest.java"));
```

- Paths.get method is shorthand for:

```
Path p4 =
   FileSystems.getDefault().getPath(“C:\users\sally");
```

File Operations

- Files class is the other primary entrypoint of the java.nio.file package
- Offers a rich set of static methods for reading, writing, and manipulating files and directories
- Files methods work on instances of Path objects

Verifying the Existence of a File or Directory

- When testing a file's existence, three results are possible:
  - file is verified to exist.
  - file is verified to not exist.
  - file's status is unknown.
    - can occur when the program does not have access to the file
- **`Files.exists(path)`**
- **`Files.notExists(path)`**
- Return value is either true or false

Checking File Accessibility

- There are three categories of accessibility that can be tested for
- **`Files.isReadable(path)`**
- **`Files.isWritable(path)`**
- **`Files.isExecutable(path)`**
- You can also check if a path is a file or a directory
- **`Files.isDirectory(path)`**
- See the JavaDoc for Files for more tests

Deleting a File or Directory

- Files class provides two deletion methods.
- ***delete(Path)*** method deletes the file or throws an exception if the deletion fails

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("no such file or directory", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format(" not empty", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

- ***deleteIfExists(Path)*** method also deletes the file, but if the file does not exist, no exception is thrown.

Copying a File or Directory

- Copy a file or directory by using:
- **`Files.copy(Path, Path, CopyOption...)`**
- Copy fails if the target file exists, unless the REPLACE_EXISTING option is specified
- Directories can be copied.
  - files inside the directory are not copied

Copying a File or Directory


```
import
java.nio.file.StandardCopyOption.*;
...
Files.copy(source, target,
REPLACE_EXISTING);
```

Moving a File or Directory

- Move a file or directory by using:
- **`Files.move(Path, Path, CopyOption...)`**
- Move fails if the target file exists, unless the REPLACE_EXISTING option is specified

```
import
java.nio.file.StandardCopyOption.*;
...
Files.move(source, target,
REPLACE_EXISTING);
```

Reading a File by Using Buffered Stream I/O

- **`Files.newBufferedReader(Path, Charset)`**
- Opens a file for reading, returning a BufferedReader that can be used to read text from a file

```
try (BufferedReader reader =
Files.newBufferedReader(file, StandardCharsets.UTF_8)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Writing a File by Using Buffered Stream I/O

- Files.newBufferedWriter(Path, Charset)
- Write to a file using a BufferedWriter.

```
String s = “Something to write to a file”;
try (BufferedWriter writer =
  Files.newBufferedWriter(file, StandardCharsets.UTF_8)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Creating Files

- Create an empty file with an initial set of attributes by using:
- **`Files.createFile(Path)`**
- If no attributes are specified the file is created with default attributes.
- If the file already exists, createFile throws an exception

```
Path file = ...;
try {
    // Create the empty file with default permissions, etc.
    Files.createFile(file);
} catch (FileAlreadyExistsException x) {
    System.err.format("file named %s" +
        " already exists%n", file);
} catch (IOException x) {
    // Some other sort of failure, such as permissions.
    System.err.format("createFile error: %s%n", x);
}
```

Exercise 15

- Remove Text (rewrite the exercise 14 and this time use BufferedWriter and BufferedReader)
- Occurrence of Each Letter

Exercise 16

(*Process scores in a text file on the Web*) Suppose that the text file on the Web http://cs.armstrong.edu/liang/data/Scores.txt contains an unspecified number of scores. Write a program that reads the scores from the file and displays their total and average. Scores are separated by blanks.