BIRZEIT UNIVERSITY

Exception Handling and Text IO

By: Mamoun Nawahdah (Ph.D.)
2020

Runtime Error?

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    System.out.println(number1 + " / " + number2 + " is " +
      (number1 / number2));
  }
}
```

2

Fix it Using an if Statement

```
import java.util.Scanner;

public class QuotientWithIf {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

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

3

Exception Handling

❖ Exception handling technique enables a method to **throw** an exception to its caller.

❖ Without this capability, a method must handle the exception or terminate the program.

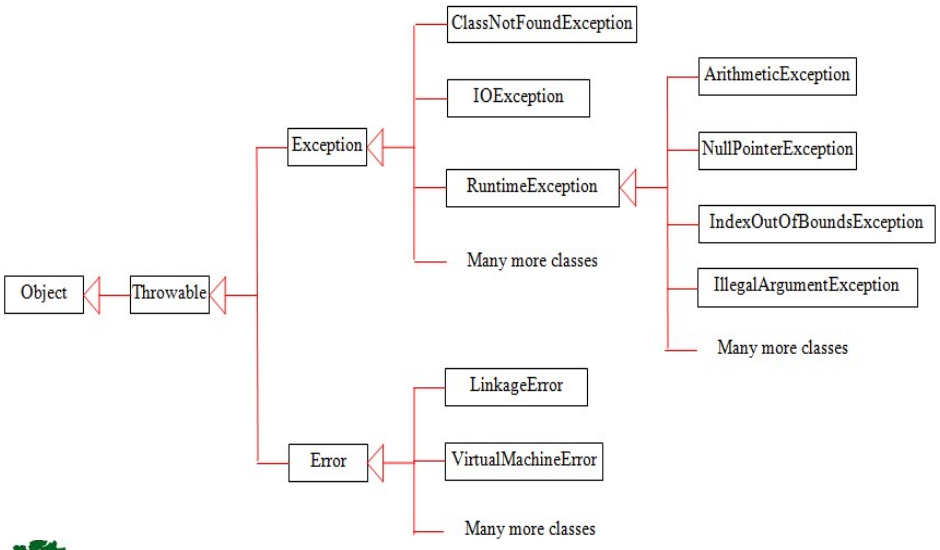
ex·cep·tion *noun* \ik-'sep-shən\

: someone or something that is different from others : someone or something that is not included

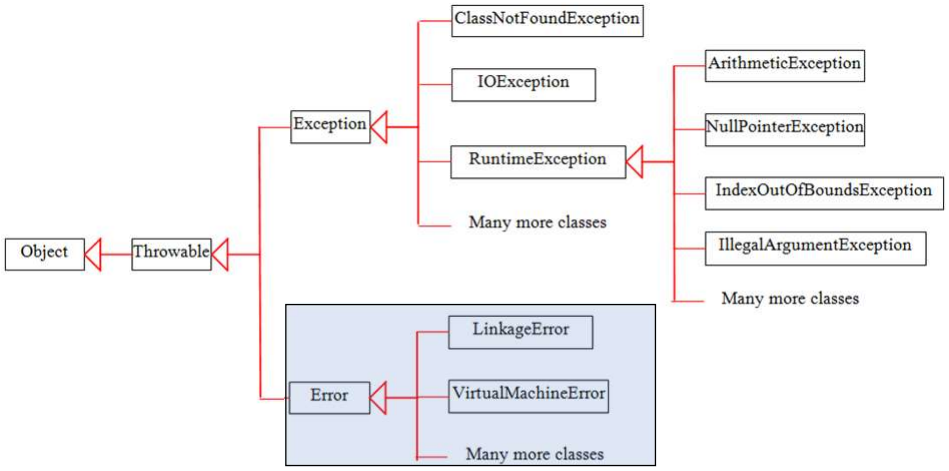
: a case where a rule does not apply

4

Exception Types

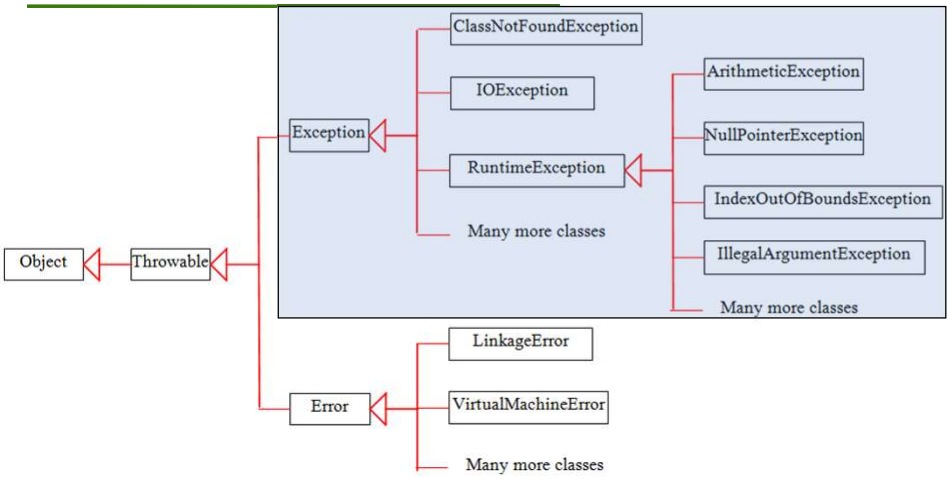


System Errors



System errors are thrown by **JVM** and represented in the **Error** class. The Error class describes internal system errors.

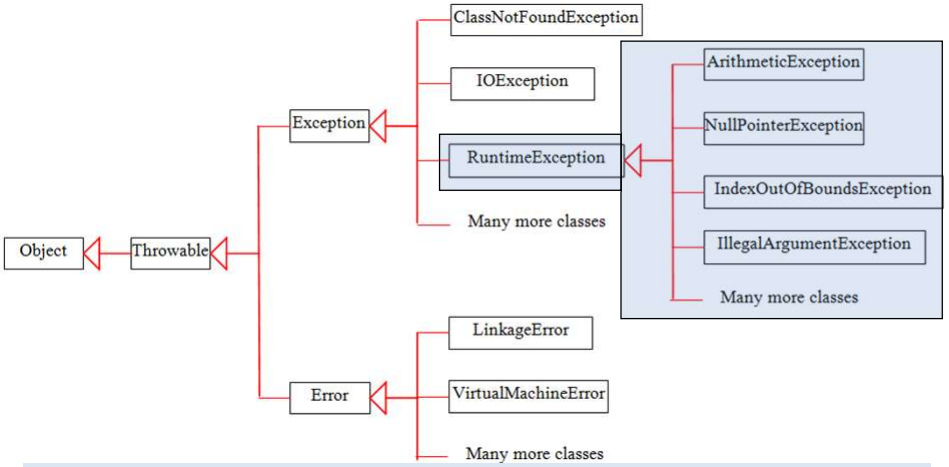
Exceptions



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

7

Runtime Exceptions



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

8

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.

9

Unchecked Exceptions

❖ In most cases, unchecked exceptions reflect programming **logic errors** that are not recoverable.

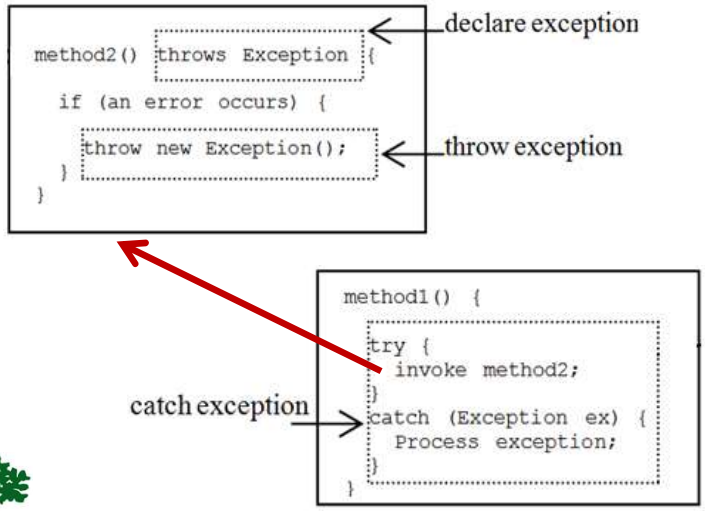
❖ For example:

- 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.

10

Declaring, Throwing, and Catching Exceptions



11

Declaring Exceptions

❖ Every method **must** state the types of checked exceptions it might **throw**.

❖ This is known as **declaring exceptions.**

```
public void x()  throws IOException

public void y()  throws IOException, OtherException
```

12

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**.

```
throw new TheException();

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

13

Throwing Exceptions Example

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

14

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;
}
```

15

Catch or Declare Checked Exceptions

Suppose **p2** is defined as follow:

```
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.
  - 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**, you have to write the code as follow:

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

17

```
 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    }
```

```
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  }
```

```
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 }
```

Rethrowing Exceptions

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

21

The finally Clause

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

22

Trace a Program Execution

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

Next statement;
```

Suppose no exceptions in the statements

23

Trace a Program Execution

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

Next statement;
```

The final block is always executed

24

Trace a Program Execution

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

Next statement;
```

Next statement in the method is executed

25

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

26

Trace a Program Execution

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

Next statement;
```

The exception is handled.

27

Trace a Program Execution

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

Next statement;
```

The final block is always executed.

28

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.

29

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.

30

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

31

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

32

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

33

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 broadcasting the errors to the calling methods.

34

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

- You should use it to deal with **unexpected** error conditions.

35

Caution!

- Do not use exception 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");
}
```

- is better to be replaced by:

```
if (refVar != null)
   System.out.println(refVar.toString());
else
   System.out.println("refVar is null");
```

36

Custom Exception

- 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 class.

37

Custom Exception Class Example

```
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 }
```

```
/** Set a new radius */
public void setRadius(double newRadius)
    throws InvalidRadiusException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new InvalidRadiusException(newRadius);
}
```

38

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.

39

File class

java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name 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 file name 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.

40

File class

`+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 path name, 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.
`+listFile(): 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 the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

41

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.

42

PrintWriter class

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."

43

Scanner class

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.

44

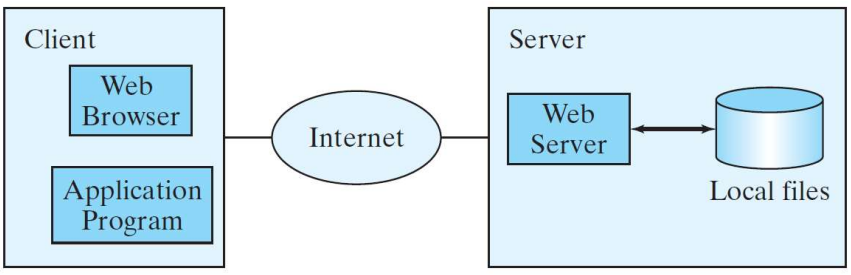
Problem: Replacing Text

- 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**

45

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.



46

Reading Data from the Web

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

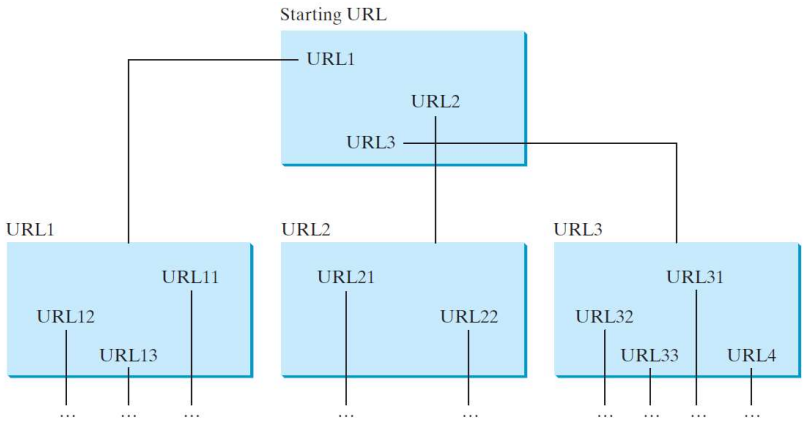
- After a **URL** object is created, you can use the **openStream()** method defined in the **URL** class to open an input stream and use this stream to create a **Scanner** object as follows:

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

47

Case Study: Web Crawler

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



48

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:

49

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;
        }
    }
}
```

50