

Assertions and Text I/O

Introduction to Programming and
Computational Problem Solving - 2

CSE 8B

Lecture 13

Announcements

- Assignment 6 is due today, 11:59 PM
- Quiz 6 is Nov 11
- Assignment 7 will be released today
 - Due Nov 16, 11:59 PM
- Educational research study
 - Nov 11, weekly survey
- Reading
 - Programming with Assertions
 - <https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>
 - Liang
 - Chapter 12

Exceptions

- Exceptions are runtime errors caused by your program and external circumstances
 - These errors can be caught and handled by your program

Exception handling

- Exception handling separates error-handling code from normal programming tasks
 - Makes programs easier to read and to modify
- The **try** block contains the code that is executed in **normal** circumstances
- The **catch** block contains the code that is executed in **exceptional** circumstances
- A method should **throw** an exception if the error needs to be handled by its caller
- **Warning: 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**

Assertions

- An assertion is a Java statement that enables you to assert an assumption about your program
- An assertion contains a Boolean expression that should be true during program execution
- Assertions can be used to assure program correctness and avoid logic errors

Declaring assertions

- An assertion is declared using the Java keyword `assert`

```
assert assertion;
```

or

```
assert assertion : detailMessage;
```

where `assertion` is a Boolean expression and `detailMessage` is a primitive-type or an Object value

Executing assertions

- When an assertion statement is executed, Java evaluates the assertion
- If it is `false`, an `AssertionError` will be thrown
- The `AssertionError` class has a no-arg constructor and seven overloaded single-argument constructors of type `int`, `long`, `float`, `double`, `boolean`, `char`, and `Object`

Executing assertions

- For the first `assert` statement with no detail message, the no-arg constructor of `AssertionError` is used
- For the second `assert` statement with a detail message, an appropriate `AssertionError` constructor is used to match the data type of the message
- Since `AssertionError` is a subclass of `Error`, when an assertion becomes `false`, the program displays a message on the console and exits

Executing assertions example

```
public class AssertionDemo {
    public static void main(String[] args) {
        int i;
        int sum = 0;
        for (i = 0; i < 10; i++) {
            sum += i;
        }
        assert i == 10;
        assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
    }
}
```

Executing assertions example

- **A best practice is to place assertions in a switch statement without a default case**

- Example

```
switch (month) {  
    case 1: ... ; break;  
    case 2: ... ; break;  
    ...  
    case 12: ... ; break;  
    default: assert false : "Invalid month: " + month  
}
```

Running programs with assertions

- By default, the assertions are disabled at runtime
- To enable them, use the switch `-enableassertions`, or `-ea` for short, as follows

```
java -ea AssertionDemo
```
- Assertions can be selectively enabled or disabled at class level or package level
- The disable switch is `-disableassertions` or `-da` for short
- For example, the following command enables assertions in package `package1` and disables assertions in class `Class1`

```
java -ea:package1 -da:Class1 AssertionDemo
```

Using exception handling or assertions

- **Assertions should not be used to replace exception handling**
- *Exception handling* deals with unusual circumstances during program execution
- *Assertions* are to assure the correctness of the program
- *Exception handling* addresses robustness
- *Assertions* address correctness
- Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks
- Assertions are checked at runtime and can be turned on or off at startup time

Using exception handling or assertions

- **Do not use assertions for argument checking in public methods**
- Valid arguments that may be passed to a public method are part of the method's contract
- The contract must always be obeyed whether assertions are enabled or disabled
 - For example, the following code in the Circle class should be rewritten using exception handling

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

Programming with assertions

- **Use assertions to reaffirm assumptions**
- This gives you more confidence to assure correctness of the program
- A common use of assertions is to replace assumptions with assertions in the code
- **A best practice is to use assertions liberally**
- Assertions are checked at runtime and can be turned on or off at startup time, *unlike exception handling*

Text I/O

- 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
 - File
 - Scanner
 - PrintWriter

Absolute file names

- Absolute file name includes full path

- Unix

- `/home/bochoa/cse8b/assignment7/Assignment7.java`

- Java string

- `String pathname =`
`"/home/bochoa/cse8b/assignment7/Assignment7.java";`

- Windows

- `C:\cse8b\assignment7\Assignment7.java`

- Java string

- `String pathname =`
`"C:\\cse8b\\assignment7\\Assignment7.java";`

Relative file names

- Relative file name includes path relative to working directory
 - For example, if you are in directory cse8b
 - Unix
 - assignment7/Assignment7.java
 - Java string
 - String pathname = "assignment7/Assignment7.java"
 - Windows
 - assignment7\Assignment7.java
 - Java string
 - String pathname = "assignment7\\Assignment7.java"

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 file name is a string
- The `File` class is a wrapper class for the file name and its directory path

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.

+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.

The File class example

```
public class TestFileClass {
    public static void main(String[] args) {
        java.io.File file = new java.io.File("image/us.gif");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("The file has " + file.length() + " bytes");
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can it be written? " + file.canWrite());
        System.out.println("Is it a directory? " + file.isDirectory());
        System.out.println("Is it a file? " + file.isFile());
        System.out.println("Is it absolute? " + file.isAbsolute());
        System.out.println("Is it hidden? " + file.isHidden());
        System.out.println("Absolute path is " +
            file.getAbsolutePath());
        System.out.println("Last modified on " +
            new java.util.Date(file.lastModified()));
    }
}
```

File 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
- Use the `Scanner` class for reading text data from a file
- Use the `PrintWriter` class for writing text data to a file

Reading data from the console

- Create a Scanner object

```
Scanner input = new Scanner(System.in);
```

- Example

```
System.out.print("Enter a double value: ");  
Scanner input = new Scanner(System.in);  
double d = input.nextDouble();
```

Reading data using Scanner

- Reading data from the console

```
Scanner input = new Scanner(System.in);
```

- Reading data from a file

```
Scanner input = new Scanner(new File(filename));
```

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 from a file

```
public class ReadData {
    public static void main(String[] args) throws Exception {
        // Create a File instance
        java.io.File file = new java.io.File("scores.txt");

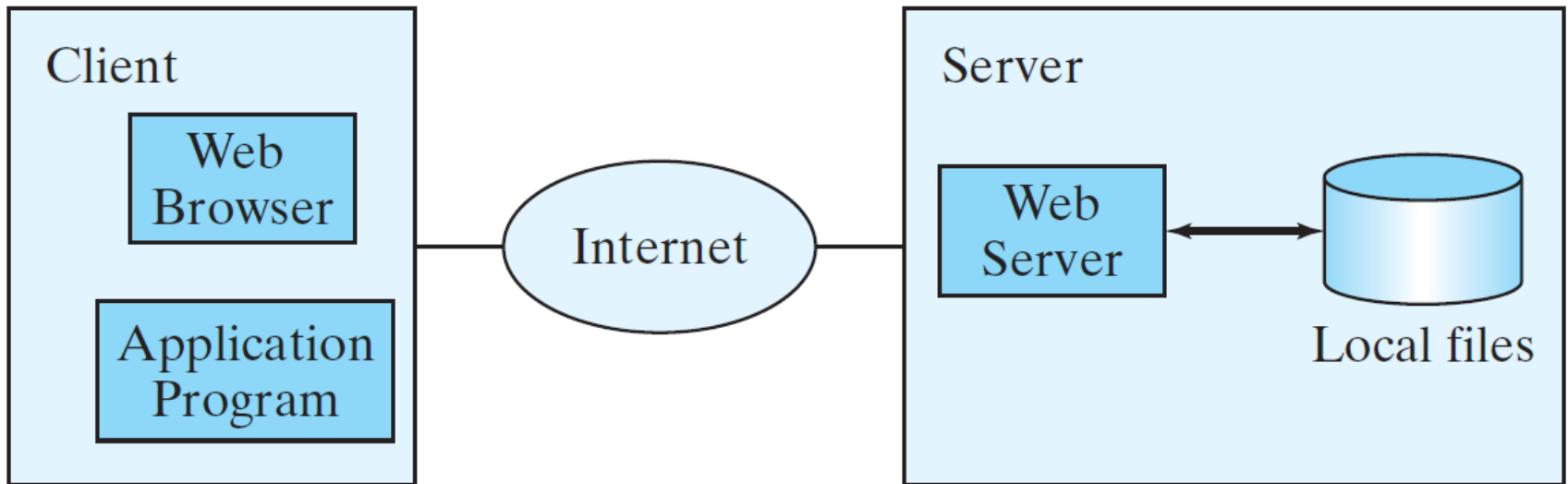
        // Create a Scanner for the file
        Scanner input = new Scanner(file);

        // Read data from a file
        while (input.hasNext()) {
            String firstName = input.next();
            String mi = input.next();
            String lastName = input.next();
            int score = input.nextInt();
            System.out.println(
                firstName + " " + mi + " " + lastName + " " + score);
        }

        // Close the file
        input.close();
    }
}
```

Reading data from the internet

- Just like you can read data from a file on the computer, you can read data from a file on the internet



Reading data from the internet

```
public class ReadFileFromURL {
    public static void main(String[] args) {
        System.out.print("Enter a URL: ");
        String urlString = new Scanner(System.in).next();

        try {
            java.net.URL url = new java.net.URL(urlString);
            int count = 0;
            Scanner input = new Scanner(url.openStream());
            while (input.hasNext()) {
                String line = input.nextLine();
                count += line.length();
            }

            System.out.println("The file size is " + count + " characters");
        }
        catch (java.net.MalformedURLException ex) {
            System.out.println("Invalid URL");
        }
        catch (java.io.IOException ex) {
            System.out.println("IO Errors");
        }
    }
}
```

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 §4.6, “Formatting Console Output and Strings.”

Writing data to a file

```
public class WriteData {
    public static void main(String[] args) throws java.io.IOException {
        java.io.File file = new java.io.File("scores.txt");
        if (file.exists()) {
            System.out.println("File already exists");
            System.exit(0);
        }

        // Create a file
        java.io.PrintWriter output = new java.io.PrintWriter(file);

        // Write formatted output to the file
        output.print("John T Smith ");
        output.println(90);
        output.print("Eric K Jones ");
        output.println(85);

        // Close the file
        output.close();
    }
}
```

Use try-with-resources syntax

- When reading or writing programmers often forget to close the file
- The try-with-resources syntax automatically closes the files
 - Write file example

```
try (  
    // Create a file  
    java.io.PrintWriter output = new java.io.PrintWriter(file);  
) {  
    // Write formatted output to the file  
    output.print("John T Smith ");  
    output.println(90);  
    output.print("Eric K Jones ");  
    output.println(85);  
}
```

File I/O example

```
public class ReplaceText {
    public static void main(String[] args) throws Exception {
        // Check command line parameter usage
        if (args.length != 4) {
            System.out.println(
                "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
            System.exit(1);
        }

        // Check if source file exists
        File sourceFile = new File(args[0]);
        if (!sourceFile.exists()) {
            System.out.println("Source file " + args[0] + " does not exist");
            System.exit(2);
        }

        // Check if target file exists
        File targetFile = new File(args[1]);
        if (targetFile.exists()) {
            System.out.println("Target file " + args[1] + " already exists");
            System.exit(3);
        }

        try (
            // Create input and output files
            Scanner input = new Scanner(sourceFile);
            PrintWriter output = new PrintWriter(targetFile);
        ) {
            while (input.hasNext()) {
                String s1 = input.nextLine();
                String s2 = s1.replaceAll(args[2], args[3]);
                output.println(s2);
            }
        }
    }
}
```

Next Lecture

- Abstract classes
- Reading
 - Liang
 - Chapter 13