CSE1710

Week 07, Lecture 14

Fall 2013 ◆ Thursday, Oct 24, 2013

YORK UNIVERSITÉ UNIVERSITY

Big Picture

The assigned coursework for today was:

- ❑ read section 4.1 “What is an Object” pp.133-136
- ❑ read section 4.2 “The Life of an Object” pp. 136-148
- ❑ review Ch 4 KC’s 1-10
- ❑ do Ch 4 RQ’s 1-23
- ❑ do Ch 4 Ex’s 4.1-4.11

YORK UNIVERSITÉ UNIVERSITY

Checklist (for next time, Lecture 15)

What you should be doing to prepare for what comes next…

- ❑ re-read section 4.2 “The Life of an Object” pp. 136-148, with a focus on 4.2.4, 4.2.5, 4.2.6
- ❑ read section 4.3 “The Object’s State” pp. 149-157
- ❑ review Ch 4 KC’s 11-16
- ❑ do Ch 4 RQ’s 23-34
- ❑ do Ch 4 Ex’s 4.12-4.22

YORK UNIVERSITÉ UNIVERSITY

The Java Class Library (JCL)

- the JCL provides developers with a set of useful facilities.
- these facilities are accessed via a set of classes (utility and non-utility), organized into packages
- E.g.,
  - `java.lang` : fundamental classes closely tied to the language and runtime system.
  - `java.io` : fundamental classes closely tied to input and output and access to the platform file system.
- Almost all of the JCL is contained in a single Java archive file called `rt.jar`
  - `rt.jar` is provided with JRE and JDK distributions

YORK UNIVERSITÉ UNIVERSITY

the *build path* vs the *class path*

- the source code of an app will depend on **services**
  - services as found in the JCL
  - services in external libraries, from other providers (e.g., JBA textbook, etc)
- these dependencies must be **resolved** at both compile time and at runtime
  - at compile time, the compiler checks the build path in order to locate any required classes on the file system
  - at run time, the JVM searches the class path to locate any required classes
- The **Java Classloader** : part of the Java Runtime Environment (JRE)
  - it loads Java classes into the Java Virtual Machine
  - usually classes are loaded **on demand**

5

YORK UNIVERSITÉ UNIVERSITY

Why are you telling me this?

Because you need to understand what the memory diagram is depicting!

BASIC Operation of the JVM:

1. Start up the class loader:
   - load the class definition that contains the main method
   - load required class definitions on demand
2. Execute bytecode:
   - execute the byte code that corresponds to the first statement of the main method
   - …execute the byte code corresponding to the next statement of the main method...
   - …repeat until final statement is reached
3. Do a tidy shutdown
   - run all shutdown hooks (close network connections, databases, etc)

6

YORK UNIVERSITÉ UNIVERSITY

… an object is an **entity** that encapsulates data representation (attributes) and computation (methods) …

- all objects of a certain type will follow the given "template" for that type: **they all will have the same set of methods and the same set of attributes, BUT:**
  - each object can have its **own values** for the given set of attributes
  - an object's methods will take into account the data that is **specific to that object**
- For instance:
  - any `Rectangle` object can have its own `width` and `height` values
  - the `getArea()` method for each `Rectangle` object will return the area that is specific to that object's `width` and `height`

YORK UNIVERSITÉ UNIVERSITY

… Non-utility classes can be thought of as factories for the creation of objects … (p. 136)

- the "factory" can be open only at runtime
- the "factory" has to be started up (the class needs to be loaded by the virtual machine)
- the objects get created by the "factory" via the use of the *constructor*
- an app can use the "factory" to crank out as many objects as desired

YORK UNIVERSITÉ UNIVERSITY

Quick Review about Fractions

- A fraction is a numerical quantity that is not necessarily a whole number.
  - integer numerator
  - non-zero integer denominator
  - types: simple $\frac{1}{2}$, complex/compound, $\frac{12\frac{3}{4}}{26}$ algebraic $\frac{1+\frac{1}{x}}{1-\frac{1}{x}}$
  - simple: proper $\frac{8}{-5}$ and improper forms $\frac{11}{4}$

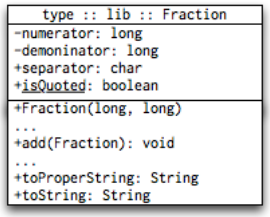
YORK UNIVERSITÉ UNIVERSITY

The `Fraction` class

- Encapsulates *simple* fractions
- Special characteristics of the `Fraction` encapsulation:
  - zero denominator permitted
  - if the fraction is negative, the negative sign is associated with numerator, not the denominator
  - fraction kept in *reduced form* at all times
    - meaning the numerator & denominator **never** have a common divisor > 1
  - supports printing both in **proper** and in **improper** formats
  - provides basic operations (add, subtract, multiply, divide)

YORK UNIVERSITÉ UNIVERSITY

The `Fraction` class



- the state of a `Fraction` object is captured by the following **four** attributes:
  - the numerator : `long`
  - the denominator : `long`
  - the character used to separate the numerator and denominator in printing : `char`
  - whether " " appears in proper format : `boolean`
- Default state:
  - `numerator = 0`
  - `denominator = 1`
  - `separator = '/'`
  - `isQuoted = true`

11

YORK UNIVERSITÉ UNIVERSITY

Exercise:

```
1  package lecture14;
2
3  import java.io.PrintStream;
4  import type.lib.Fraction;
5
6  public class Example01 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 1);
13         output.printf("%s%n", f1.toString());
14         output.printf("%s%n", f1.toProperString());
15
16         Fraction f2;
17         f2 = new Fraction(4, 2);
18         output.printf("%s%n", f2.toString());
19         output.printf("%s%n", f2.toProperString());
20     }
21
22 }
```

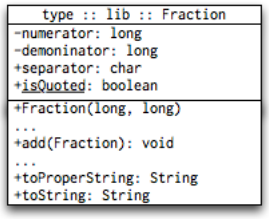
Can you predict the output?

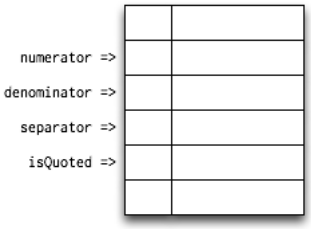
12

YORK UNIVERSITÉ UNIVERSITY

Exercise:

For the class `Fraction`,
assign the zero-offset addresses in the same way the compiler would

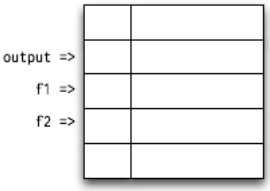




YORK UNIVERSITÉ UNIVERSITY

Exercise:

For the app `Example01`,
assign the zero-offset addresses in the same way the compiler would



YORK UNIVERSITÉ UNIVERSITY

Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example01`

(up to the point in time when the bytecode corresponding to line 11 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
4  import type.lib.Fraction;
5
6  public class Example01 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 1);
13         output.printf("%s%n", f1.toString());
14         output.printf("%s%n", f1.toProperString());
15
16         Fraction f2;
17         f2 = new Fraction(4, 2);
18         output.printf("%s%n", f2.toString());
19         output.printf("%s%n", f2.toProperString());
20     }
21
22 }
```


YORK UNIVERSITÉ UNIVERSITY

Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example01`

(up to the point in time when the bytecode corresponding to line 12 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
4  import type.lib.Fraction;
5
6  public class Example01 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 1);
13         output.printf("%s%n", f1.toString());
14         output.printf("%s%n", f1.toProperString());
15
16         Fraction f2;
17         f2 = new Fraction(4, 2);
18         output.printf("%s%n", f2.toString());
19         output.printf("%s%n", f2.toProperString());
20     }
21
22 }
```


YORK UNIVERSITÉ UNIVERSITY

Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example01`

(up to the point in time when the bytecode corresponding to line 16 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
4  import type.lib.Fraction;
5
6  public class Example01 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 1);
13         output.printf("%s%n", f1.toString());
14         output.printf("%s%n", f1.toProperString());
15
16         Fraction f2;
17         f2 = new Fraction(4, 2);
18         output.printf("%s%n", f2.toString());
19         output.printf("%s%n", f2.toProperString());
20     }
21
22 }
```


Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example01`

(up to the point in time when the bytecode corresponding to line 17 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
4  import type.lib.Fraction;
5
6  public class Example01 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 1);
13         output.printf("%s%n", f1.toString());
14         output.printf("%s%n", f1.toProperString());
15
16         Fraction f2;
17         f2 = new Fraction(4, 2);
18         output.printf("%s%n", f2.toString());
19         output.printf("%s%n", f2.toProperString());
20     }
21
22 }
```

Exercise:

Predict the output of the app `Example02`

```
1  package lecture14;
2
3  import java.io.PrintStream;
5
6  public class Example02 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 3);
13         output.printf("%s%n", f1.toString());
14
15         Fraction f2;
16         f2 = f1;
17         f1.setNumerator(4);
18         f2.setDenominator(5);
19         output.printf("%s%n", f2.toString());
20         output.printf("%s%n", f2.toString());
21     }
22 }
```


YORK UNIVERSITÉ UNIVERSITY

Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example02`
(up to the point in time when the bytecode corresponding to line 11 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
5
6  public class Example02 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 3);
13         output.printf("%s%n", f1.toString());
14
15         Fraction f2;
16         f2 = f1;
17         f1.setNumerator(4);
18         f2.setDenominator(5);
19         output.printf("%s%n", f2.toString());
20         output.printf("%s%n", f2.toString());
21     }
22 }
```


YORK UNIVERSITÉ UNIVERSITY

Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example02`
(up to the point in time when the bytecode corresponding to line 12 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
5
6  public class Example02 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 3);
13         output.printf("%s%n", f1.toString());
14
15         Fraction f2;
16         f2 = f1;
17         f1.setNumerator(4);
18         f2.setDenominator(5);
19         output.printf("%s%n", f2.toString());
20         output.printf("%s%n", f2.toString());
21     }
22 }
```

21

YORK UNIVERSITÉ UNIVERSITY

Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example02`
(up to the point in time when the bytecode corresponding to line 15 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
5
6  public class Example02 {
7
8      public static void main(String[] args) {
9          PrintStream output = System.out;
10
11         Fraction f1;
12         f1 = new Fraction(2, 3);
13         output.printf("%s%n", f1.toString());
14
15         Fraction f2;
16         f2 = f1;
17         f1.setNumerator(4);
18         f2.setDenominator(5);
19         output.printf("%s%n", f2.toString());
20         output.printf("%s%n", f2.toString());
21     }
22 }
```

22

YORK UNIVERSITÉ UNIVERSITY

Exercise:

Draw a memory diagram to illustrate the contents of memory for the app `Example02`
(up to the point in time when the bytecode corresponding to line 16 is invoked)

```
1  package lecture14;
2
3  import java.io.PrintStream;
4
5  public class Example02 {
6
7      public static void main(String[] args) {
8          PrintStream output = System.out;
9
10         Fraction f1;
11         f1 = new Fraction(2, 3);
12         output.printf("%s%n", f1.toString());
13
14         Fraction f2;
15         f2 = f1;
16         f1.setNumerator(4);
17         f2.setDenominator(5);
18         output.printf("%s%n", f2.toString());
19         output.printf("%s%n", f2.toString());
20     }
21 }
```


YORK UNIVERSITÉ UNIVERSITY

Key Concept 4.3

To create an instance of a class at runtime, design your source code as follows:

> Import the class. Use the `new` keyword in conjunction with the invocation of the class's constructor. These statements, when compiled into bytecode, will take care of instance creation.

The invocation of the constructor, (as accomplished via the corresponding bytecode, at runtime) will cause the instance to be created and the object's initial state to be set.

YORK UNIVERSITÉ UNIVERSITY

Key Concept 4.4

The default constructor is one that does not take any parameters. Using it leads to an instance with a default initial state. Nondefault constructors do take parameters and allow you to customize the initial state of the created instance.

YORK UNIVERSITÉ UNIVERSITY

Key Concept 4.5

The created instance, also called an object, occupies a block in memory reserved for it. The address of that block is the object's "identity" (location in memory). To be able to refer to this object, we declare a variable, called the object reference, to be of the class type and assign its value to be the object's memory location.

The object reference is thus a "pointer" that points at the object. It is sometimes referred to as the object's handle.

YORK UNIVERSITÉ UNIVERSITY