CSE1710

Week 07, Lecture 13

Fall 2013 ◆ Tuesday, Oct 22, 2013

Big Picture

The assigned reading was for today:

- ❑ read section 3.3 “General Characteristics of Utility Classes”
- ❑ review Ch 3 KC’s 15-18
- ❑ do Ch 3 RQ’s 26-30
- ❑ do Ch 3 Exercises 3.1-3.22 (+ Lab L3.2 “A Software Project”)

Checklist (for next time, Lecture 14)

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

- ❑ 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

Review Questions

RQ.26 When a class is compiled, how does the compiler know where in memory the class will be loaded?

In reading questions, ask yourself:

1. what are the **presuppositions** of the question?
2. Are these **presuppositions** indeed true?

Presupposition: something that is assumed to hold true at the outset, but that is not stated overtly

Review Questions

RQ.26 When a class is compiled, how does the compiler know where in memory the class will be loaded?

Presupposition: that the compiler knows where in memory the class will be loaded

But the compiler **does not know** in advance where in memory a given class will be loaded. It ***cannot*** know, since there are too many variables.

The question might instead be:
how are the addresses of class features assigned at compile time and at run time?

YORK UNIVERSITÉ UNIVERSITY

5

Class Features: Addressing

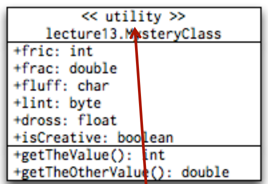
- A class' features are its *attributes* and *methods* **[Fig 2.9, p. 81]**
- A class' attributes are its variables
  - they can be private or public
  - class variables that are public are **fields**
- At compile time, a starting address of zero is assumed. Every feature within a class is given an consecutive address, relative this zero-offset
- At runtime (class loading), the zero offset is replaced with a non-zero offset. The addresses of all of the features are shifted up.

YORK UNIVERSITÉ UNIVERSITY

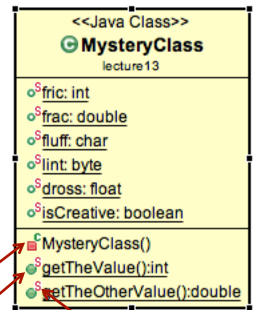
6

UM Diagram for `MysteryClass`

manually prepared



automatically generated using Eclipse plug-in



stereotype << `utility` >> means no publically-available constructor, all features are static

red square means "this feature is private"

green circle means "this feature is public"

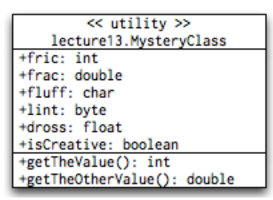
"S" means static feature

YORK UNIVERSITÉ UNIVERSITY

Example:

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

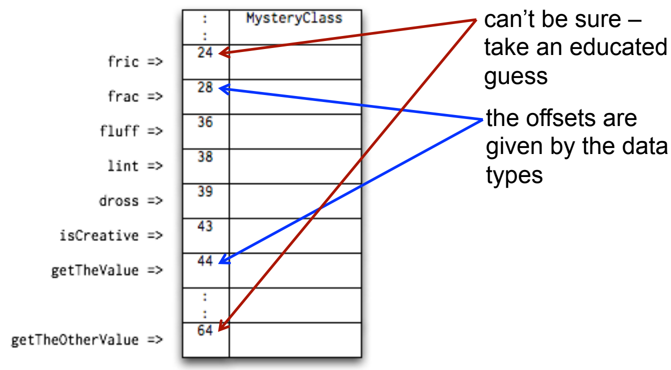
you may assume all features are shown in the UML class diagram



fric =>		
frac =>		
fluff =>		
lint =>		
dross =>		
isCreative =>		
getTheValue =>		
getTheOtherValue =>		

YORK UNIVERSITÉ UNIVERSITY

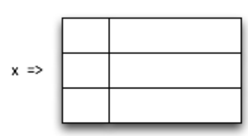
Sample Solution



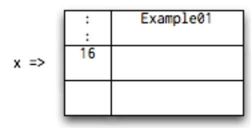
Example:

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

```
package lecture13;

public class Example01 {

    public static void main(String[] args) {
        float x = 16.4f;
        MysteryClass.dross = x;
    }

}
```



Sample Solution



11

YORK UNIVERSITÉ UNIVERSITY

Example:

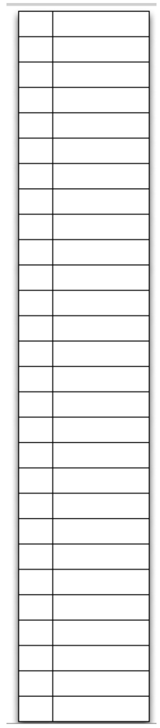
Draw a memory diagram to illustrate the contents of memory for this app

(up to the point in time when the bytecode corresponding to line 7 is invoked, but the app has not yet terminated)

```
1  package lecture13;
2
3  public class Example01 {
4
5      public static void main(String[] args) {
6          float x = 16.4f;
7          MysteryClass.dross = x;
8      }
9
10 }
```

12

YORK UNIVERSITÉ UNIVERSITY



	200	Example01
x =>	216	16.4f
	600	MysteryClass
fric =>	624	
frac =>	628	
fluff =>	636	
lint =>	638	
dross =>	639	16.4f
isCreative =>	643	
getTheValue =>	644	
	: :	
getTheOtherValue =>	664	

Pros and Cons

Utility	Non-Utility
less versatile	more versatile
API is simpler (no constructor section, cannot create instances)	API is more complex
at runtime, class definition is loaded into memory	at runtime, class definition is loaded into memory, plus an object is created each time the class is instantiated
all attributes are static	attributes are static or non-static
all methods are static	methods are static or non-static
suitable for services that do not need to store information about *state*	suitable for services that need to store information about *state*

About the class `Integer`

```
package lecture13;

import java.io.PrintStream;

public class Example02 {

    public static void main(String[] args) {
        PrintStream output = System.out;
        int x1 = 67;
        String x2 = "67";
        Integer y1 = new Integer(x1);
        Integer y2 = Integer.parseInt(x2);
        Integer y3 = x1;
        int x3 = y3;
        output.printf("%d%n", x1);
        output.printf("%s%n", x2);
        output.printf("%d%n", x3);
        output.printf("%d%n", y1);
        output.printf("%d%n", y2);
        output.printf("%d%n", y3);
    }
}
```

three ways to get an `Integer` *object*

auto-boxing

auto-unboxing

Static features in class `Integer`

```java
package lecture13;

import java.io.PrintStream;

public class Example03 {

    public static void main(String[] args) {
        PrintStream output = System.out;
        String x2 = "1";
        Integer y2 = Integer.parseInt(x2);
        // here we see a static attribute of the class Integer
        output.printf("max int:\t %20d%n", Integer.MAX_VALUE);
        // here we demonstrate the wrap-around property of integers
        int result = Integer.MAX_VALUE + y2;
        output.printf("result:\t %20d%n", result);
    }
}
```

YORK UNIVERSITÉ UNIVERSITY

A non-static method in `Integer`

```java
package lecture13;

import java.io.PrintStream;

public class Example04 {

    public static void main(String[] args) {
        PrintStream output = System.out;
        String x1 = "87";
        int x2 = 87;
        Integer y1 = Integer.parseInt(x1);
        int result = y1.compareTo(x2);
        output.printf("result:\t %2d%n", result);
    }
}
```

YORK UNIVERSITÉ UNIVERSITY

Input Validation

Suppose you are expecting a numeric value that obeys some sort of condition. For instance:

```
enter a non-zero positive integer:
```

How can we perform **validation**?

19

Input Validation

Validation Options	
#1	let the app crash or make it crash
#2	stop the app (but **not** by crashing) and tell the user the reason
#3	inform the user of the problem and re-prompt

20

Input Validation

we know how to do this!

Validation Options		
Exception-Based	#1	let the app crash or make it crash
Message	#2	stop the app (but **not** by crashing) and tell the user the reason
Friendly	#3	inform the user of the problem and re-prompt

requires loops [Ch 5]

requires selection (`if` statement) [Ch 5]

Input Validation

Suppose you are expecting a numeric value that obeys some sort of condition. For instance:

`enter a non-zero positive integer:`

How can we perform **validation**?

Scenarios:

1. user enters something other than an `int`
   - we can take advantage of the services provided by `Scanner` or `Integer`
2. user enters an `int`, but it is zero or negative
   - we can take advantage of the services provided by `Scanner` or `Integer`

Exception-Based Validation

1. Need to validate the ***type*** of the user input
2. Need to validate the ***value*** of the user input

Exception-Based Validation

[Approach #1] To validate the ***type*** of the user input, use the services of `Scanner`

`public int nextInt()`

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:
the `int` scanned from the input

Throws:
`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range
`NoSuchElementException` - if input is exhausted
`IllegalStateException` - if this scanner is closed

Example05

```java
package lecture13;

import java.io.PrintStream;

public class Example05 {

    public static void main(String[] args) {
        PrintStream output = System.out;
        Scanner input = new Scanner(System.in);
        final String PROMPT = "enter a non-zero positive integer:";
        output.printf("%s%n", PROMPT);
        int userValue = input.nextInt();
        output.printf("inputted value: %d%n", userValue);
    }
}
```


YORK UNIVERSITÉ UNIVERSITY

Exception-Based Validation

[Approach #2] To validate the ***type*** of the user input, use the services of `Integer`

parseInt

```
public static int parseInt(String s)
                throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:
`s` - a `String` containing the `int` representation to be parsed

Returns:
the integer value represented by the argument in decimal.

Throws:
`NumberFormatException` - if the string does not contain a parsable integer.

YORK UNIVERSITÉ UNIVERSITY

Example06

```java
package lecture13;

import java.io.PrintStream;

public class Example06 {

    public static void main(String[] args) {
        PrintStream output = System.out;
        Scanner input = new Scanner(System.in);
        final String PROMPT = "enter a non-zero positive integer:";
        output.printf("%s%n", PROMPT);
        String userInput = input.nextLine();
        int userValue = Integer.parseInt(userInput);
        output.printf("inputted value: %d%n", userValue);
    }
}
```


Exception-Based Validation

To validate the ***value*** of the user input, construct a `boolean` expression:

```
boolean isValid = userValue > 0;
```

The conditionally trigger a runtime error using the services of `ToolBox`

```
        final String MSG = “Amount was not non-zero
positive value”;

        ToolBox.crash(!isValid, MSG);
```

```java
package lecture13;

import java.io.PrintStream;

public class Example07 {

    public static void main(String[] args) {
        PrintStream output = System.out;
        Scanner input = new Scanner(System.in);
        final String PROMPT = "enter a non-zero positive integer:";
        output.printf("%s%n", PROMPT);
        String userInput = input.nextLine();
        int userValue = Integer.parseInt(userInput);
        final String MSG = "Amount was not non-zero positive value!";
        boolean isValue = userValue > 0;
        ToolBox.crash(!isValue, MSG);
        output.printf("inputted value: %d%n", userValue);
    }
}
```


YORK U UNIVERSITÉ UNIVERSITY