ETH zürich

Assignment 2

Felix Friedrich, Fabian Stutz, Lars Widmer

TA lecture, Informatics II D-BAUG

March 11, 2014

Outline

Strange Effects

Can you explain?

1. `if (str.charAt(i) == 'e') ...`
   works **well**!
2. `if (str=="*") ...`
   does **not** work! Instead we have to use:
   `if (str.equals("*")) ...`

Value Types

Question

Can you give examples of **value types** rsp. **base types**?

Value Types: Examples

- `char`: Letters
- `int`: Integer numbers
- `long`: Long integer numbers
- `float`: Floating point numbers
- `double`: Double precision floating point numbers
- `short`: Short integer numbers

Recognizing Value Types

Question

How can you recognize **value types**?

Recognizing Value Types

- Eclipse typesets the value types in bold red.
- You don't need the `new` statement to create an `int` variable.
- Variables of value types never offer methods.
- They are all predefined, we can't create our own ones.
- They are spelled all lowercase.
- The value types are small.

✓ That's why we usually don't call value types to be classes.

✓ And variables of base types we usually don't call objects.

Reference Types

Question

Can you give examples of **reference types**?

Reference Types: Examples

- `String`
- `Array`: lists of elements
- Any `Classes`
- ...

Recognizing Reference Types

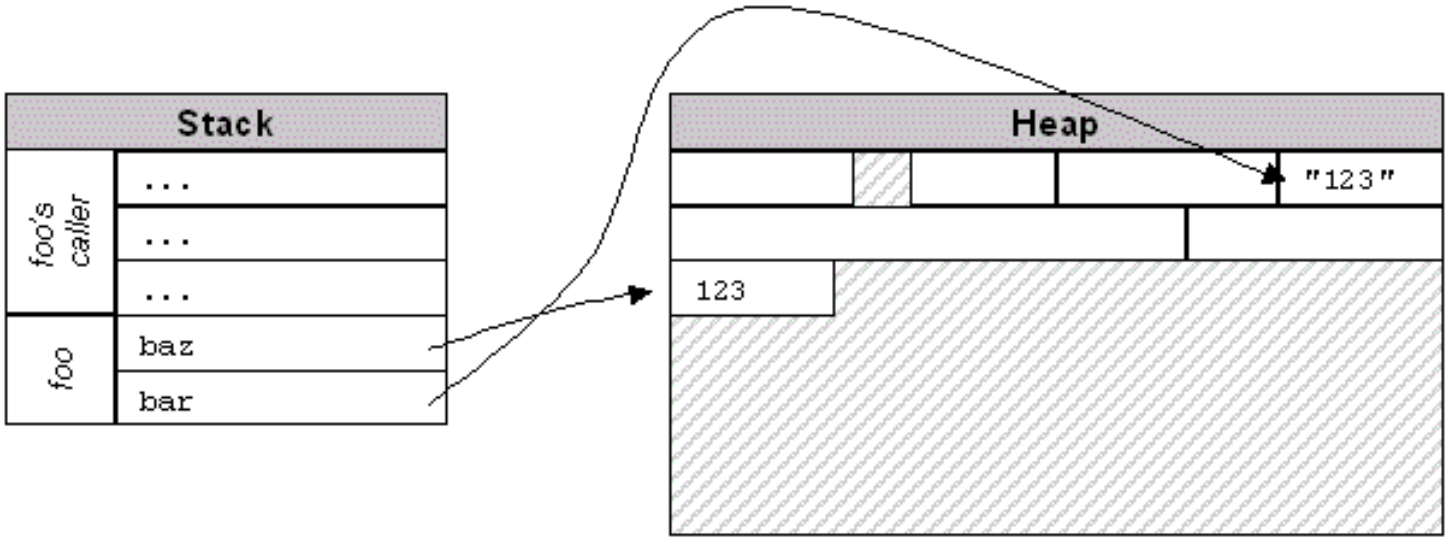
Question

How can you recognize **reference types**?

Recognizing Reference Types

- Eclipse doesn't highlight reference types.
- They start with an uppercase letter.
- You always need the `new` statement to create an object.
  Exception: If you write a String like `"Hello"`, java creates a new String behind the scenes for you. The rule still holds, but you can't see the `new` statemant in the code.
- Objects often/usually offer methods.
- We **can** create our own classes.

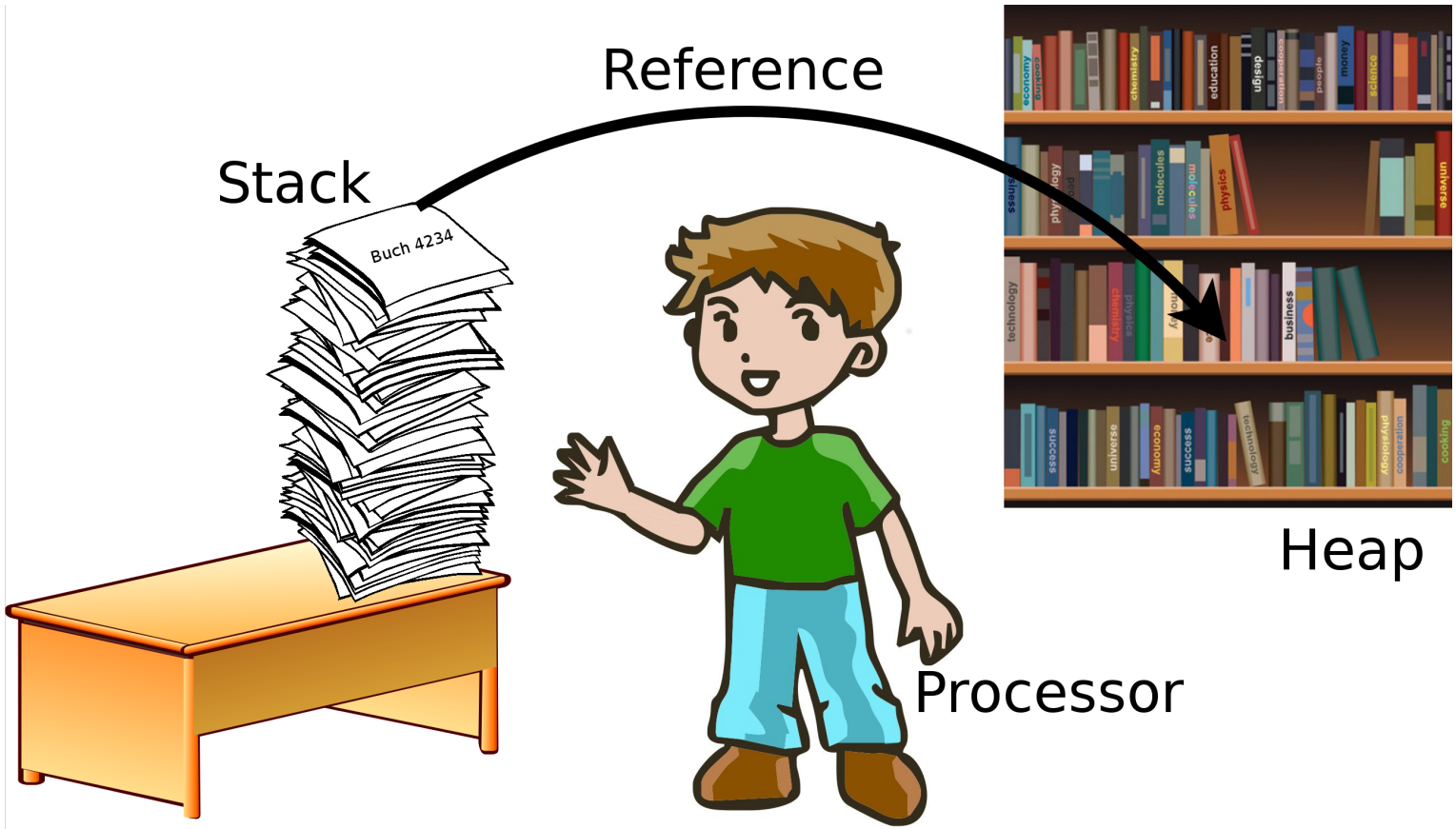
References are Pointers



Source: kdgregory.com

- The arrows depict the pointers.

If you're a Processor ...



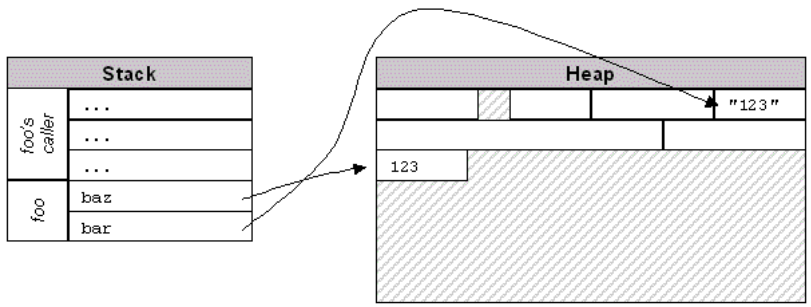
If you're a Processor ...

As an analogy you may think of your room being a **computer** while you as a **processor** are doing your homework which is the **program**:

- Your **stack** is a pile of sheets of paper on your desk.
- An **object** could be a book.
- If you pile 100 books on your desk, you can't work anymore.
- Therefore only single sheets of papers go to your **stack**.
- Books you place in your bookshelf.
- So to speak, your bookshelf is your **heap** memory.
- You may place a note on piece of paper about which book your using for what purpose. This note is a **reference**. The **reference** is kept in the **stack**, but it points to the **heap**.

References are Pointers

- Every program has its own stack.
- Every method has its part of the stack.
- Every variable of a method is located in its stackframe.
- Since larger items (like objects) can't be put on the stack, they are stored on the heap.



References as Values

- As a matter of fact, objects have a **value** as well!
- The **value** is the **reference**.
- The reference points to the address of the object. The memory is addressed with numbers. That's why the number in the reference is often referred as a **pointer**.
- If we have `TestClass a = new TestClass();` an object is created and placed in memory. In the variable `a` the **address** (**reference**) of the object is stored.
- Whereas with value types, there is no referenceing needed. It's just the value and only the value, as the name suggests.

Reference & Value Types

1. That's why the statement `(str=="*")` compares just the two references (addresses!).
2. Since the two objects are created seperately, they'll never reside at the same location.
3. Hence `(str=="*")` always returns `false`.
4. That's a **very common pitfall**, be aware of it! Some people complain that java should handle this different, but if it did it won't be consequent!

Outline

Vector-Matrix Multiplication

The aim is to show how a vector converges to a fixed value, while it's being multiplied iterativly by the same matrix.

$$V_1 = V_0 \times M$$
$$V_2 = V_1 \times M$$
$$V_3 = V_2 \times M$$
$$\dots$$
$$V_{n+1} = V_n \times M$$

For increasing n the following should hold:
$V_n \to V_\infty$

Single Step

$$V_0 = \begin{pmatrix} 1 & 2 \end{pmatrix}$$

$$M = \begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix}$$

$$V_1 = V_0 \times M$$

$$V_1 = \begin{pmatrix} 1 \times 3 + 2 \times 5 & 1 \times 4 + 2 \times 6 \end{pmatrix}$$

$$V_1 = \begin{pmatrix} 13 & 16 \end{pmatrix}$$

Convergence

$$M = \begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix}$$

$$V_0 = \begin{pmatrix} 1 & 2 \end{pmatrix}, V_1 = \begin{pmatrix} 13 & 16 \end{pmatrix} \ldots$$

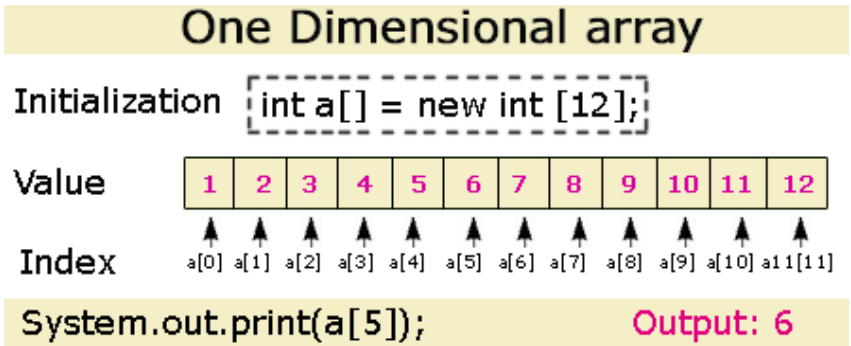
Open Question

Will this example converge?
. . . Play around with it in exercise 2.

Outline

Arrays

Same like in Pascal an Array is used to store a list of some elements. All elements have the same type.



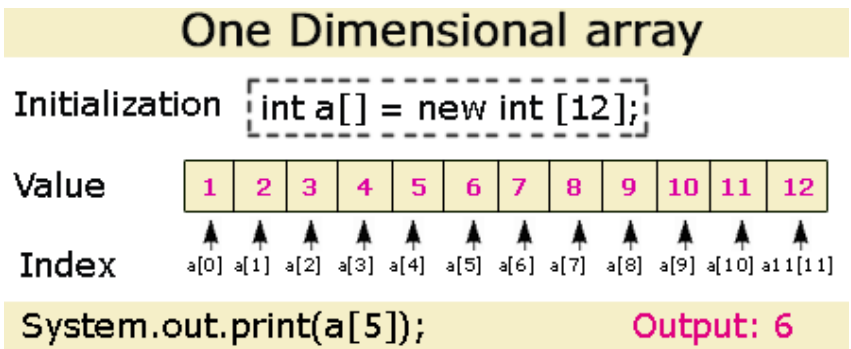
Source: http://www.roseindia.net

Arrays

You can think of an Array like a predefined class.

But . . .

1. What does that mean?
2. How do we use it?



Source: http://www.roseindia.net

Arrays

1. It's a reference type (not a value/base type like `int`).
2. This means an Array object resides in memory.
3. This can be seen because we need the `new` keyword to create an object of type Array.
4. Example: `int[] list = new int[20];`
   (size 20: indices 0..19)
5. The line above contains actually two statements:
   - `int[] list;` (Declaration)
   - `list = new int[20];` (Creation + Initialisation)
6. Since `int` is a value type, we don't need a new statement for every single value in the Array.
7. This would be different e. g. if we create an Array of `String`.

Outline

Reversing, Given Code

```
1  import java.util.Scanner;
2  public class Reverse {
3      public static void main(String[] args) {
4          String str = "";
5          String rev = "";
6          Scanner sc = new Scanner(System.in);
7          System.out.print("Please enter a word: ");
8          str = sc.next();
9          System.out.println(str);
10         for (int i=0; i<str.length(); ++i)
11             rev = str.charAt(i) + rev;
12         System.out.println(rev);
13     }
14 }
```

Reversing, Computations

```
1 for (int i=0; i<str.length(); ++i)
2     rev = rev + str.charAt(i);
```

Riddle:

1. Who can see the difference?
2. What's the result?

```
1 for (int i=0; i<str.length(); ++i)
2     rev = str.charAt(i) + rev;
```

Reversing, Wrong Computation

```
1 for (int i=0; i<str.length(); ++i)
2     rev = rev + str.charAt(i);
```

This code goes through the word, from the beginning to the end. One after another, it concatinates the letters to the result variable `rev`. As a matter of fact, the result will be the same as the input.

Reversing, Correct Computation

```
1 for (int i=0; i<str.length(); ++i)
2     rev = str.charAt(i) + rev;
```

This piece of code does it different: Still it goes through the word, from the beginning to the end. One after another, it concatinates the letters to the result variable `rev`. **But** it adds them at the **beginning** of the result string. Therefore the reversing of the string results.

Outline

Count Letters: Full Method

```
1  import java.util.Scanner;
2  public class CountLetter {
3      public static void main(String[] args) {
4          int count = 0;
5          Scanner sc = new Scanner(System.in);
6          System.out.print("Please enter a word: ");
7          String str = sc.next();
8          for (int i=0; i<str.length(); ++i)
9              if (str.charAt(i) == 'e')
10                 ++count;
11         System.out.println(count);
12     }
13 }
```

Count Letters: Two Different For-Loops

```
1 for (int i=0; i<str.length(); ++i)
2     rev = str.charAt(i) + rev;
```

Question

Compare the two snipplets. What are the differences?

```
1 for (int i=0; i<str.length(); ++i)
2     if (str.charAt(i) == 'e')
3         ++count;
```

Count Letters: The Real Counting

```
1 for (int i=0; i<str.length(); ++i)
2     if (str.charAt(i) == 'e')
3         ++count;
```

. . . Still it goes through the word, from the beginning to the end. But whenever it finds the letter “e” it just increments the value of count.

Outline

Jumbled Strings

Arniodccg to a recheaesrr at Cmbragide Ueytrvsiin, it dosen't mettar in what oedrr the leertts in a wrod aer, the only itpramnot thing is taht the frist and lsat ltteer be at the rhigt plcae. The rest can be a toatl mses and you can sltil read it whtuoit pberolm. This is bscueae the hamun mnid deos not read erevy letter by istlef but the word as a wolhe.

Original Text

According to a researcher at Cambridge University, it doesn't matter in what order the letters in a word are, the only important thing is that the first and last letter be at the right place. The rest can be a total mess and you can still read it without problem. This is because the human mind does not read every letter by itself but the word as a whole.

Jumbling: Outer Program Structure

```
1 import java.util.LinkedList;
2 import java.util.Random;
3 import java.util.Scanner;
4 import java.lang.Character;
5
6 public class MixMiddle {
7     // ...
8 }
```

A `LinkedList` is a data structure similar to an `Array`. But it allows to remove arbitrary elements, even from the middle. This feature simplifies the implementation of MixMiddle.

Jumbling: Question

Method `jumble`

Looking at the method `jumble`:
Can you see distinct phases, the code goes through?
We go through them one by one . . .

Jumbling: All Code

```
1  private static String jumble(String str) {
2      int len = str.length();
3      if (len < 4) return str;
4      Random rand = new Random();
5      LinkedList<Character> letters
6              = new LinkedList<Character>();
7      for (int i=1; i<len-1; ++i)
8          letters.add(str.charAt(i));
9      String mix = "" + str.charAt(0);
10     for (int i=0; i<len-2; ++i)
11         mix += letters.remove(
12                 rand.nextInt(letters.size()));
13     mix += str.charAt(len-1);
14     return mix;
15 }
```

Jumbling: Return Statements

Question

What's the goal of the two `return`-statements below?

```
1 private static String jumble(String str) {
2     int len = str.length();
3     if (len < 4) return str;
4     // ...
5     return mix;
6 }
```

Jumbling: Ending Method, Returning Results

```
1 private static String jumble(String str) {
2     int len = str.length();
3     if (len < 4) return str;
4     // ...
5     return mix;
6 }
```

The `return`-statement ends a method immediately. It doesn't have to be at the end of the method. That's quick but not very beautiful, since it can be misleading. That's exactly why you should know about it.

Jumbling: LinkedList ...

Question

What does the following code snipplet?

```
1       LinkedList<Character> letters
2               = new LinkedList<Character>();
3       for (int i=1; i<len-1; ++i)
4           letters.add(str.charAt(i));
```

Jumbling: Copy String

```
1       LinkedList<Character> letters
2               = new LinkedList<Character>();
3       for (int i=1; i<len-1; ++i)
4           letters.add(str.charAt(i));
```

Here we convert the word into a `LinkedList` of letters.
But we don't insert the **first** and the **last** letter.

Jumbling: For-Loop

Question

And what's going one here?

```
1       String mix = "" + str.charAt(0);
2       for (int i=0; i<len-2; ++i)
3           mix += letters.remove(
4                   rand.nextInt(letters.size()));
5       mix += str.charAt(len-1);
```

Jumbling: Drawing Letters Randomly

```
1       String mix = "" + str.charAt(0);
2       for (int i=0; i<len-2; ++i)
3           mix += letters.remove(
4                   rand.nextInt(letters.size()));
5       mix += str.charAt(len-1);
```

The result variable `res` is introduced. For the first and the last letter we use the original letters, since we don't want them to be "jumbled".
In between we randomly draw a letter from the `LinkedList` and concatinate it to the result variable. The `LinkedList` ensures that we **can't draw** the same letter **twice**.

Jumbling: Overview

The method which jumbles the letters goes through the following steps:

1. Check if the word is longer than 4 characters.
2. Copy the word into a `LinkedList`.
3. Keep the first letter of the word as the first letter of the result.
4. For every letter between the first and the last pick a random one and concatinate it to the result.
5. Set the last letter of the word as the last letter of the result.
6. Return result

Jumble Main: Full Code

```
1  public static void main(String[] args) {
2      System.out.println("Enter * to exit");
3      Scanner sc= new Scanner(System.in);
4      String str = "-";
5      while (!str.contains("*") && sc.hasNext()) {
6          str = sc.next();
7          System.out.print(jumble(str) + " ");
8      }
9      System.out.println("\n... program done");
10 }
```

Jumble Main: Questions

Tricky Questions

1. Why do we check if there is a `"*"` in the string?
2. Could use `str.equals("*")` instead of `str.contains("*")`?
3. Could we use `(str=="*")` instead of `str.equals("*")`?

```
1       while (!str.contains("*") && sc.hasNext()) {
2           str = sc.next();
3           System.out.print(jumble(str) + "␣");
4       }
```

Jumble Main: Checking for a Star

```
1       while (!str.contains("*") && sc.hasNext()) {
2           str = sc.next();
3           System.out.print(jumble(str) + "␣");
4       }
```

We check for a `"*"` to provide an easy way to end the program when typing the words directly (**not** using a file for input with a pipe: `cat text.txt | java MixMiddle`).

Jumble Main: Comparing Strings

```
1       while (!str.contains("*") && sc.hasNext()) {
2           str = sc.next();
3           System.out.print(jumble(str) + "␣");
4       }
```

Again: `String` is a `reference type` similar to `Array`s and `class`es in java but different from `int` or `char` (value types). The statement `(str=="*")` compares just the two references. The reference states were the `String`-object is located in memory. Since the two objects are created seperately, they'll never reside at the same location. Hence `(str=="*")` always returns false. That's a **very common pitfall**, be aware of it!

Outline

Binary Numbers

Calculate the “normal” value (to the base of 10) of following binary numbers:

1. 001
2. 010
3. 100
4. 011
5. 111
6. 0100 0000 0000
7. 1000 0000 0000

Hex Numbers

Calculate the “normal” value (to the base of 10) of following hex numbers:

1. *A*
2. *B*
3. *F*
4. *FF*
5. 14
6. 21
7. *A*0

Questions?

Please

- Questions?
- Feedback?
- Wishes?
- Remarks?
- ...

Good Success!