Lesson 31: Variables and Data Types #2 (W09D3)

Balboa High School

Michael Ferraro

October 15, 2015

Do Now

Attempt to answer from memory, recording your responses.

---

1. An `int` can hold a number as pos. or neg. as $\pm$ _____ .
2. A `float` uses ____ bytes, while a `double` uses twice as many.
3. When Java assigns a value to a variable's memory address, the first step it performs is _____ of the RHS of the statement.
4. The default value for an instance variable of type `int` is ____ . The same is not true, however, for local variables.
5. What step(s) does the `new` operator perform?[1]

[1] From the required reading, Litvin §§5.1–5.3.

Aim

Students will learn about `String` objects (initialization, concatenation, and escape sequences), constants, and variable scope.

Strings in Java

- A `String` can be thought of as a set of characters in a specific order

[2] common abbreviation in academia referring to the Latin "confer," which translates to "compare" or "consult."

Strings in Java

- A `String` can be thought of as a set of characters in a specific order
  - "AbCdE$" is a `String` of 6 characters

[2] common abbreviation in academia referring to the Latin "confer," which translates to "compare" or "consult."

Strings in Java

- A `String` can be thought of as a set of characters in a specific order
  - "AbCdE$" is a `String` of 6 characters
  - " " is a `String` of 1 character, a space

[2] common abbreviation in academia referring to the Latin "confer," which translates to "compare" or "consult."

Strings in Java

- A `String` can be thought of as a set of characters in a specific order
  - "AbCdE$" is a `String` of 6 characters
  - " " is a `String` of 1 character, a space
  - "" is the empty `String`, which has 0 characters

[2] common abbreviation in academia referring to the Latin "confer," which translates to "compare" or "consult."

Strings in Java

- A `String` can be thought of as a set of characters in a specific order
  - "AbCdE$" is a `String` of 6 characters
  - " " is a `String` of 1 character, a space
  - "" is the empty `String`, which has 0 characters
- They're objects in Java; cf.[2] primitives

[2] common abbreviation in academia referring to the Latin "confer," which translates to "compare" or "consult."

Strings in Java

- A `String` can be thought of as a set of characters in a specific order
  - "AbCdE$" is a `String` of 6 characters
  - " " is a `String` of 1 character, a space
  - "" is the empty `String`, which has 0 characters
- They're objects in Java; cf.[2] primitives
- The `String` class is part of the `java.lang` package; its API from the AP subset can be found here

[2] common abbreviation in academia referring to the Latin "confer," which translates to "compare" or "consult."

Initializing a `String` Object

- Creating a new `String`:

```
String str = new String();
```

Initializing a `String` Object

- Creating a new `String`:

```
String str = new String();
```

- Using the constructor to set a new `String`'s value:

```
String str = new String("Hello there!");
```

Initializing a `String` Object

- Creating a new `String`:

```
String str = new String();
```

- Using the constructor to set a new `String`'s value:

```
String str = new String("Hello there!");
```

- Shorthand for the last example:

```
String str = "Hello there!"
```

String Concatenation

- *Concatenate* — to link together

String Concatenation

- *Concatenate* — to link together
- You've used this many times: the + operator! E.g.,

```
int i = 6;
System.out.println("The value of i is " + i + ".");
```

String Concatenation

- *Concatenate* — to link together
- You've used this many times: the + operator! E.g.,

```
int i = 6;
System.out.println("The value of i is " + i + ".");
```

- In the example above, two `String`s and an `int` were *concatenated* together to form one `String`:

String Concatenation

- *Concatenate* — to link together
- You've used this many times: the + operator! E.g.,

```
int i = 6;
System.out.println("The value of i is " + i + ".");
```

- In the example above, two `String`s and an `int` were *concatenated* together to form one `String`:
  - A *string literal*: `"The value of i is "`

String Concatenation

- *Concatenate* — to link together
- You've used this many times: the + operator! E.g.,

```
int i = 6;
System.out.println("The value of i is " + i + ".");
```

- In the example above, two `String`s and an `int` were *concatenated* together to form one `String`:
  - A *string literal*: `"The value of i is "`
  - An `int`: `i` (Java replaced it w/its value, 6)

String Concatenation

- *Concatenate* — to link together
- You've used this many times: the + operator! E.g.,

```
int i = 6;
System.out.println("The value of i is " + i + ".");
```

- In the example above, two `String`s and an `int` were *concatenated* together to form one `String`:
  - A *string literal*: `"The value of i is "`
  - An `int`: `i` (Java replaced it w/its value, 6)
  - Another *string literal*: `"."`

String Concatenation

- *Concatenate* — to link together
- You've used this many times: the + operator! E.g.,

```
int i = 6;
System.out.println("The value of i is " + i + ".");
```

- In the example above, two `String`s and an `int` were *concatenated* together to form one `String`:
  - A *string literal*: `"The value of i is "`
  - An `int`: `i` (Java replaced it w/its value, 6)
  - Another *string literal*: `"."`
  - Resulting `String`: `"The value of i is 6."`

Coming Later

- We will work with `Strings` at a greater depth in Ch. 8
- Want a preview? Check out the `String` API from the APCS Java Subset

Constants

There are two kinds of constants (cf. *variables*) in Java:

- Literal Constants
- Symbolic Constants

Literal Constants

```
int t = -7;              // -7 is a constant

char c = 'q';            // q is a constant

double d = 3.1415926     // 3.1415926 is a constant

String str = "abc";      // abc is a constant
                         // (aka a string literal)
```

Symbolic Constants

- Wouldn't it be painful for a programmer to have to type 3.1415926 every time (s)he needs to use that value?

[3] *im-* ≈ not, *mutate* ≈ change, *-able* ≈ able to;
∴ **immutable** means *cannot change*

Symbolic Constants

- Wouldn't it be painful for a programmer to have to type 3.1415926 every time (s)he needs to use that value?
- Enter the *symbolic constant*:
  `final double PI = 3.1415926;`

  The keyword `final` makes the `double` called `PI` *immutable*[3]

[3] *im-* ≈ not, *mutate* ≈ change, *-able* ≈ able to;
∴ **immutable** means *cannot change*

Symbolic Constants

- Wouldn't it be painful for a programmer to have to type 3.1415926 every time (s)he needs to use that value?
- Enter the *symbolic constant*:
  `final double PI = 3.1415926;`

  The keyword `final` makes the `double` called `PI` *immutable*[3]
- Ex: If you wanted *a* to equal 4.1415926,
  `double a = PI + 1;`

[3] *im-* ≈ not, *mutate* ≈ change, *-able* ≈ able to;
∴ **immutable** means *cannot change*

Symbolic Constants

- Wouldn't it be painful for a programmer to have to type 3.1415926 every time (s)he needs to use that value?
- Enter the *symbolic constant*:
  `final double PI = 3.1415926;`

  The keyword `final` makes the `double` called `PI` *immutable*[3]
- Ex: If you wanted *a* to equal 4.1415926,
  `double a = PI + 1;`
- Naming Convention: CAPITAL LETTERS for the names of constants

[3] *im-* ≈ not, *mutate* ≈ change, *-able* ≈ able to;
∴ **immutable** means *cannot change*

What's the use?

- Use of *symbolic constants* in programming is a best practice

What's the use?

- Use of *symbolic constants* in programming is a best practice
- Let's say that you use `PI` all over your program. Later, you decide that you need to use a more specific value for `PI`.

What's the use?

- Use of *symbolic constants* in programming is a best practice
- Let's say that you use `PI` all over your program. Later, you decide that you need to use a more specific value for `PI`.
  - using *symbolic constants*: simply change your definition of `PI`:
    `final double PI = 3.14159265358979;`

What's the use?

- Use of *symbolic constants* in programming is a best practice
- Let's say that you use `PI` all over your program. Later, you decide that you need to use a more specific value for `PI`.
  - using *symbolic constants*: simply change your definition of `PI`: `final double PI = 3.14159265358979;`
  - without *symbolic constants*: change the value of π **everywhere** you typed it; for that matter, **everywhere everyone on your programming team typed it!**

What's the use?

- Use of *symbolic constants* in programming is a best practice
- Let's say that you use `PI` all over your program. Later, you decide that you need to use a more specific value for `PI`.
  - using *symbolic constants*: simply change your definition of `PI`: `final double PI = 3.14159265358979;`
  - without *symbolic constants*: change the value of π **everywhere** you typed it; for that matter, **everywhere everyone on your programming team typed it!**
- Simply put: Symbolic constants are good for program maintenance.

`String` Escape Sequences

- What if you'd like to include the double quote character — `"` — in a `String`?

`String` Escape Sequences

- What if you'd like to include the double quote character — `"` — in a `String`?
- `String a =`
  `"He said "no."";` ← confuses the Java compiler!

`String` Escape Sequences

- What if you'd like to include the double quote character — `"` — in a `String`?
- `String a = "He said "no."";` ← confuses the Java compiler!
- Solution: Place a backslash — `\` — before the character you want; this *escapes* the character from the compiler's search for a `String` delimiter, the double quote

```
String a = "He said \"no.\"";
```

More Escape Sequences

Strings may have these other escape sequences embedded within them:

- `\n` — newline character, advances cursor to beginning of next line (akin to pressing [ENTER])

  Ex:

  ```
  String q = "Line 1,\nLine 2.";
  System.out.println(q);
  ```

  produces this output:

  ```
  Line 1,
  Line 2.
  ```

More Escape Sequences

Strings may have these other escape sequences embedded within them:

- \t — tab character, advances cursor to next tab stop; useful for aligning columns for multiple lines of output (see this in PS #5)

More Escape Sequences

Strings may have these other escape sequences embedded within them:

- \\ — produces a backslash character in a `String`
- \' — produces a single quote character (apostrophe)

  Ex:

```
String q = "this is a \\ and a \' :)";
System.out.println(q);
```

  produces this output:

```
this is a \ and a ' :)
```

Variable Scope

Let's read the code on the next slide and predict how it will behave. . .

Variable Scope

```
public class Scope {
    public static void main(String[] args) {

        int i = 10;

        while ( i > 5 ) {
            System.out.println("I can see that i is " + i);
            i--;
        }

        if ( i == 5 ) {
            int j = 6;
            System.out.println("j is currently " + j);
        }

        System.out.println("outside the if block, j is " + j);
    }
}
```

Variable Scope

- Download the source for `Scope.java` from here and import into a new project called `Lesson31`
- Figure out *why* there's a problem.
- Try to fix the program so it behaves as we believe it *should*.

Classwork & HW

- PS #5, §§1-3, inclusive