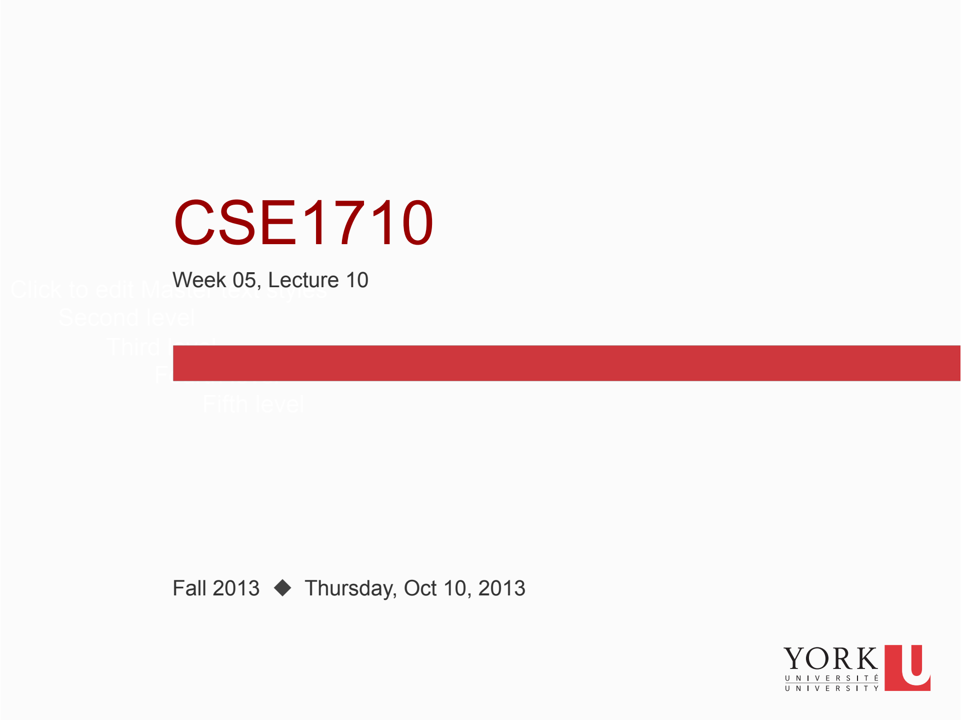


CSE1710

Week 05, Lecture 10

Click to edit Master text styles

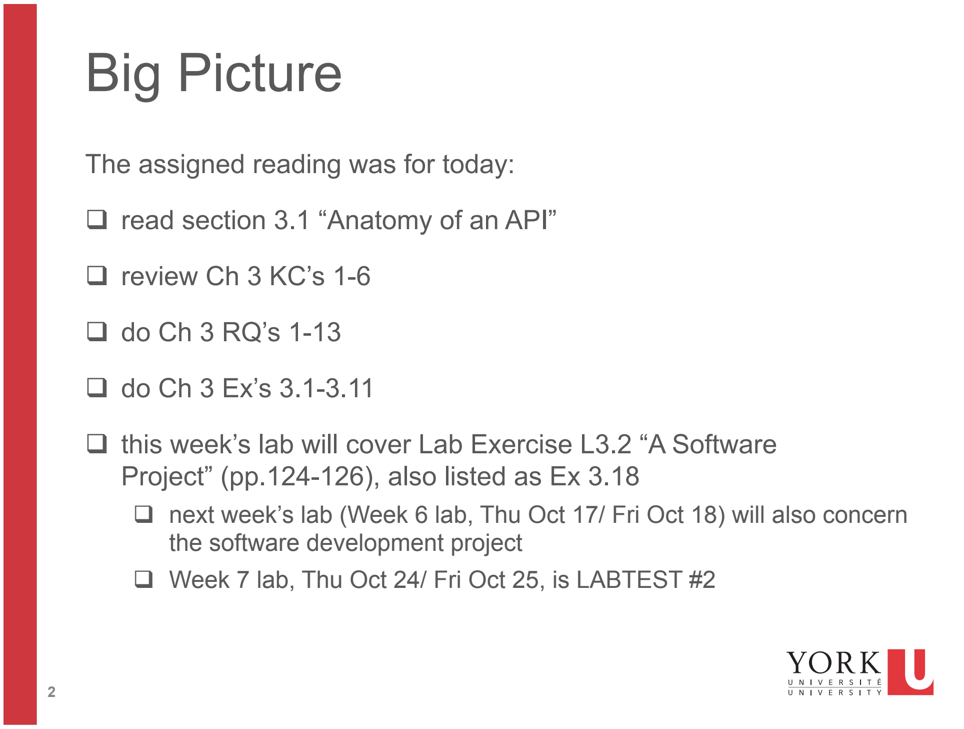
Second level

Third

F

Fifth level

Fall 2013 ◆ Thursday, Oct 10, 2013



Big Picture

The assigned reading was for today:

- ❑ read section 3.1 “Anatomy of an API”
- ❑ review Ch 3 KC’s 1-6
- ❑ do Ch 3 RQ’s 1-13
- ❑ do Ch 3 Ex’s 3.1-3.11
- ❑ this week’s lab will cover Lab Exercise L3.2 “A Software Project” (pp.124-126), also listed as Ex 3.18
  - ❑ next week’s lab (Week 6 lab, Thu Oct 17/ Fri Oct 18) will also concern the software development project
  - ❑ Week 7 lab, Thu Oct 24/ Fri Oct 25, is LABTEST #2

Checklist (for Today)

What we are reinforcing with the exercises this class…

- □ ability to identify overloaded methods in an API
- □ ability to determine bindings
- □ ability to recognize implications of “passing by value” in a practical way
    - □ probing what is meant by the value of a primitive variable vs the value of a non-primitive variable

Checklist (for next time, Lecture 11)

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

- □ read section 3.2 “A Development Walk-Through”
- □ review Ch 3 KC’s 7-14
- □ do Ch 3 RQ’s 13-25
- □ do Ch 3 Ex’s 3.12-3.16
- □ last week’s lab covered Lab Exercise L3.2 “A Software Project” (pp. 124-126), also listed as Ex 3.18

Exercise 3.3

Visit the Java API. Two of the classes are called `Date`.

(a) How can there be two classes with the same name?

(b) If such a name were referenced in a program, how would the compiler know which one to bind with?

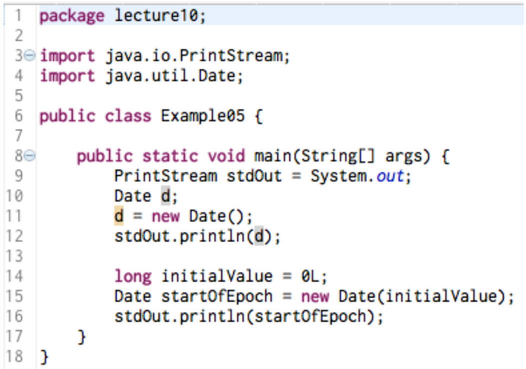
(c) Is the `compareTo` method in `java.util.Date` overloaded?

A side note about `Date`

- We will use the `Date` class in the package `java.util`
  - ignore the other `Date` class in the package `java.sql`
- The class provides services for representing and working with points in time
  - relative to the Gregorian calendar and a time zone
  - e.g., `Wed Oct 09 14:21:01 EDT 2013`
- The date is encapsulated as a `long` value
  - represents the number of milliseconds that have elapsed since **unix epoch**.
  - Unix epoch is represented by `0L` and corresponds to: `Jan 01 00:00:00 UTC 1970`

A side note about `Date`

Predict the output of this app.



```
package lecture10;

import java.io.PrintStream;
import java.util.Date;

public class Example05 {

    public static void main(String[] args) {
        PrintStream stdOut = System.out;
        Date d;
        d = new Date();
        stdOut.println(d);

        long initialValue = 0L;
        Date startOfEpoch = new Date(initialValue);
        stdOut.println(startOfEpoch);
    }
}
```

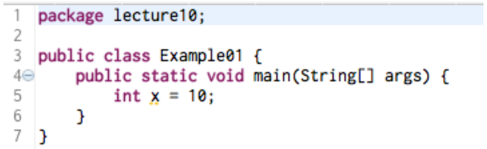
Parameters Passed by Value

Key Concept #4

Parameters in Java are **passed by value**. This means only their values are sent to the invoked method. Other languages provide **pass by reference** in which the address is sent. Passing by value is safer because methods cannot change variable local to the calling program.

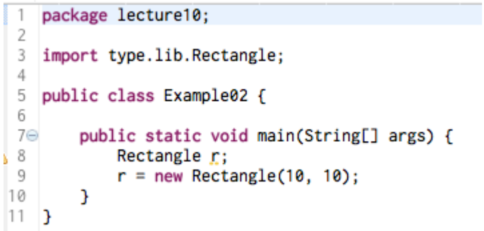
Recap: What is meant by the **value** of a variable?

Exercise: draw a diagram that illustrates the contents of memory for each of the following apps.



```
package lecture10;

public class Example01 {
    public static void main(String[] args) {
        int x = 10;
    }
}
```

*What is the **value** of* `x`*?*



```
package lecture10;

import type.lib.Rectangle;

public class Example02 {

    public static void main(String[] args) {
        Rectangle r;
        r = new Rectangle(10, 10);
    }
}
```

*What is the **value** of* `r`*?*

Recap: What is meant by the **value** of a variable?

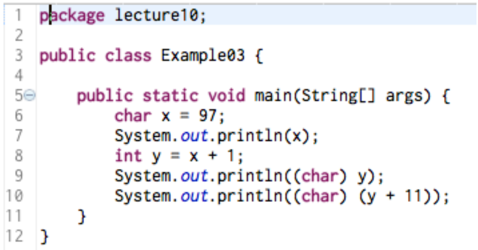
The take away point is as follows:

the value of a non-primitive variable is an address

Variable	Characteristics of its value
primitive	• a set of using 0's and 1's that corresponds to a **numerical or boolean value** • the numerical or boolean value is determined according to the relevant representation scheme (e.g., `int`, `long`, `double`, … etc)
non-primitive	• a set of using 0's and 1's that corresponds to a **memory location** (address) • the only type of address that is valid is the starting byte of an object in runtime memory that has the same (compatible) type as the declaration of the variable • e.g., the statement `Rectangle r;` means that the variable `r` holds an address and that address must be the starting byte of a `Rectangle` object

A side note about `char`

Predict the outcome:

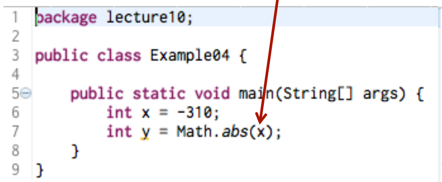


```
package lecture10;

public class Example03 {

    public static void main(String[] args) {
        char x = 97;
        System.out.println(x);
        int y = x + 1;
        System.out.println((char) y);
        System.out.println((char) (y + 11));
    }
}
```

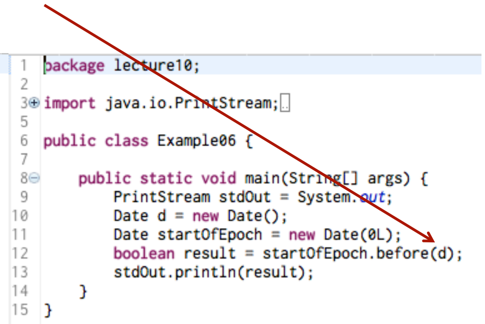
Do you see how `char` is actually a numerical type?

Parameter Passing

For each of these method invocations, identify the **value** that is passed.

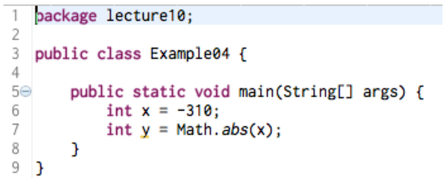


```
package lecture10;

public class Example04 {

    public static void main(String[] args) {
        int x = -310;
        int y = Math.abs(x);
    }
}
```



```
package lecture10;

import java.io.PrintStream;

public class Example06 {

    public static void main(String[] args) {
        PrintStream stdOut = System.out;
        Date d = new Date();
        Date startOfEpoch = new Date(0L);
        boolean result = startOfEpoch.before(d);
        stdOut.println(result);
    }
}
```

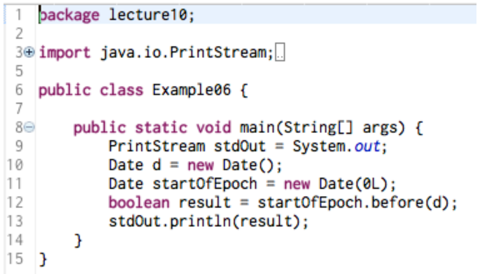
Parameters Passed by Value



```
package lecture10;

public class Example04 {

    public static void main(String[] args) {
        int x = -310;
        int y = Math.abs(x);
    }
}
```

Could the method `abs(int)` change the **value** of `x`?

Parameters Passed by Value



```
package lecture10;

import java.io.PrintStream;□

public class Example06 {

    public static void main(String[] args) {
        PrintStream stdOut = System.out;
        Date d = new Date();
        Date startOfEpoch = new Date(0L);
        boolean result = startOfEpoch.before(d);
        stdOut.println(result);
    }
}
```

Could the method `before(Date)` change the **value** of `d`?

Could the method `before(Date)` change the **state** of `d`?

Exercise 3.10

Bind the invocation `Math.round(2.5)`

(a) Is the method `round` overloaded? (examine API of `Math` class)

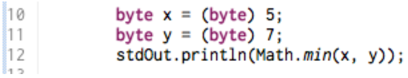
Write a short program that proves your binding

How could you prove the following:
(c1) the invocation **does not** bind with `round(float)`

[hint: use the compiler's type checking]

Exercise 3.11

Consider the fragment



```
byte x = (byte) 5;
byte y = (byte) 7;
stdOut.println(Math.min(x, y));
```

(a) Is the method `min` overloaded? (examine API of `Math` class)
(b) With which method will the compiler bind this invocation of `min`?

How could you prove the following:
(c1) the invocation **does not** bind with `min(long, long)`
(c2) the invocation **does not** bind with `min(double, double)`

[hint: use the compiler's type checking]