

Lecture 8

**Reviewing Methods;
Wrapper Classes**

Announcements for This Lecture

Readings

- Sections 2.5 and 5.1
- PLive 5-1 and 5-2



Assignments

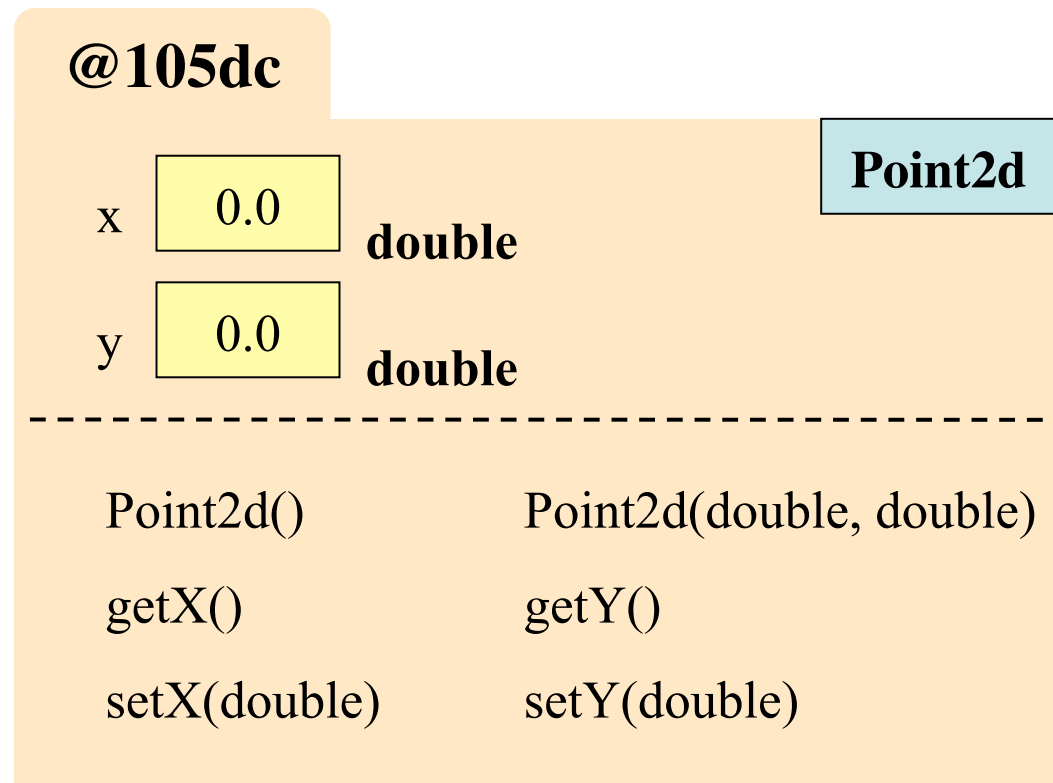
- Assignment 2 due Tuesday
 - Turn in during class
 - Not accepting on CMS
- Assignment 1 being graded
 - Graded so far: 128 of 195
 - Keep revising until Tuesday
- Assignment 3 will go up this weekend (to give 1.5 weeks)
 - Extension of Assignment 1

How Do We Do Assignment 2?

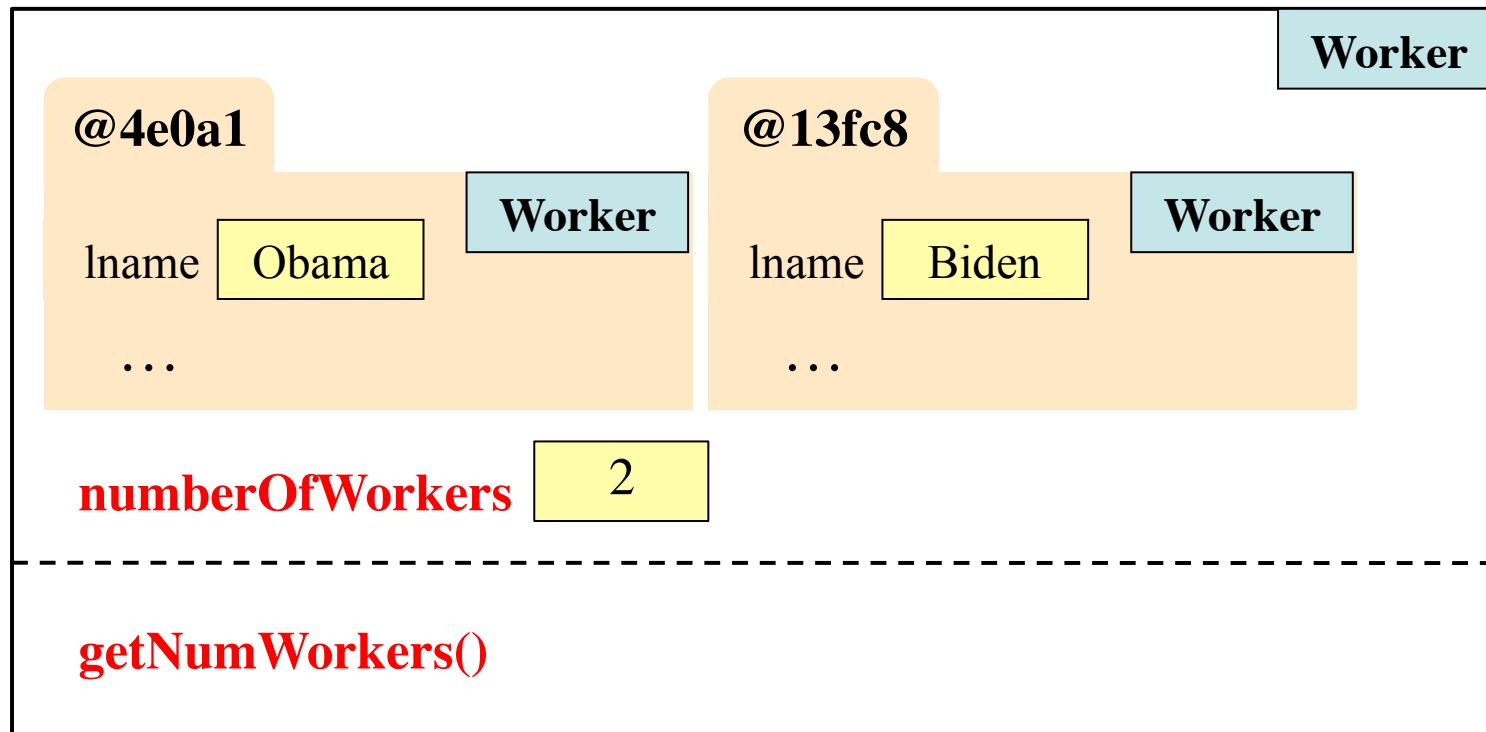
Objects (Folders) and Variables

p @105dc Point2d

q @105dc Point2d

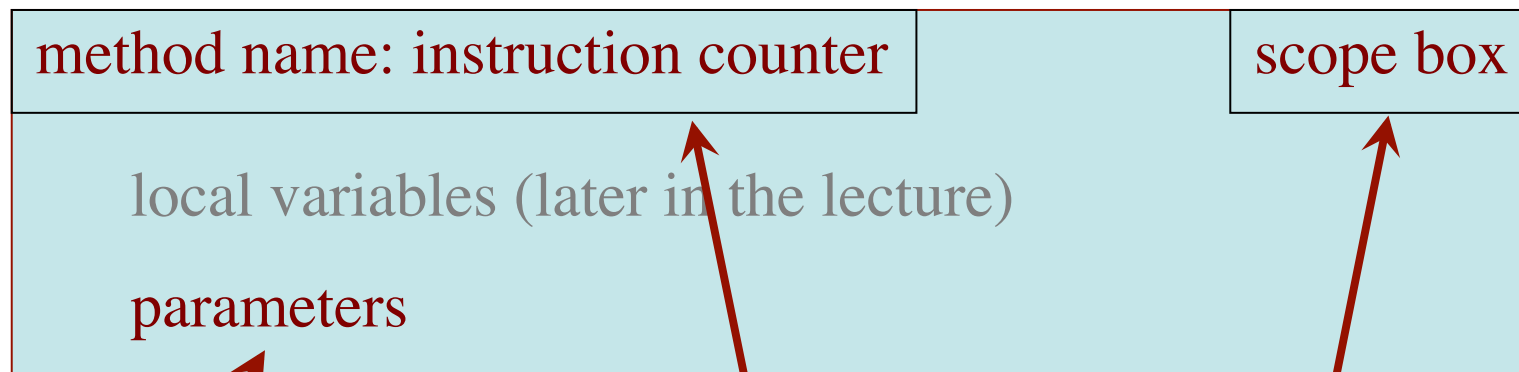


Classes (File Drawers)



Method Frames

- The formal representation of a method call



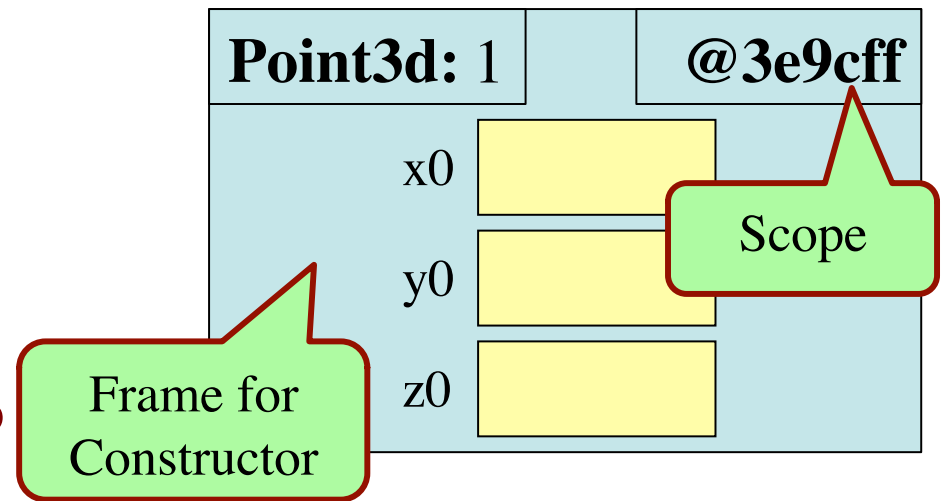
Draw parameters
as variables
(e.g. boxes)

- Number of the statement in method body to execute next
- **Starts with 1**
- Helps you keep track of where you are

- Object (folder) name if an instance method
- Class if a static method

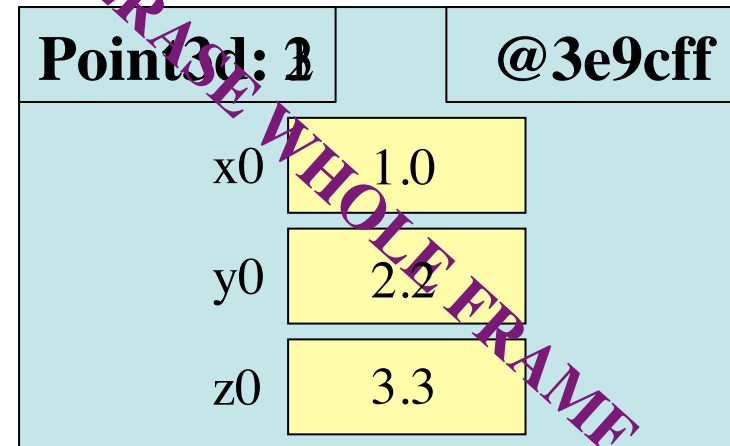
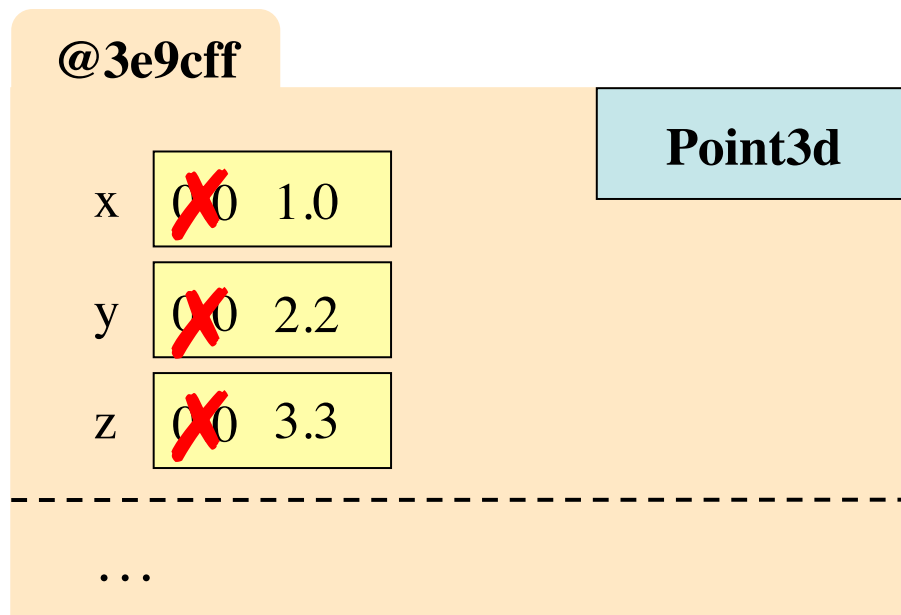
Constructors are Instance Methods

1. Make a new object (folder)
 - Java gives the folder a name
 - All fields are default (0 or null)
2. Draw a frame for the call
3. Assign the argument value to the parameter (in frame)
4. Execute the method body
 - Look for variables in the frame
 - Execute statements to initialize the fields to non-default values
 - Give the folder name as the result
5. Erase the frame for the call



```
public Point3d( double x0,  
               double y0,  
               double z0) {  
  
    x = x0;  
    y = y0;  
    z = z0;  
  
}
```

Example: `p = new Point3d(1.0, 2.2, 3.3);`



```
public Point3d( double x0,
                double y0,
                double z0) {
    x = x0;
    y = y0;
    z = z0;
}
```


Types of Method Calls

With a Dot (.)

- Instance method call
 - Method applied to an object
 - `<object>.<method-call>`
 - **Example:** `p.getX()`
- Static method call
 - Definition in file drawer
 - `<class>.<method-call>`
 - **Ex:** `Integer.parseInt("123")`

Without a Dot (.)

- Helper method call
 - Call is executed inside body of another method
 - Both are in same class file
- Scope box contains scope of the method that called it
- **Example:** `firstName(s)`

Helper Methods

```
/**
 * Yields: copy of s but in the form
 * <last-name>, <first-name>
 * Precondition: s is in the form
 * <first-name> <last-name>
 * with one blank between names
 */
public static String
lastNameFirst(String s) {
    String first = firstName(s);
    // Find the last name
    // Put together with comma
    return first; // Stub
}
```

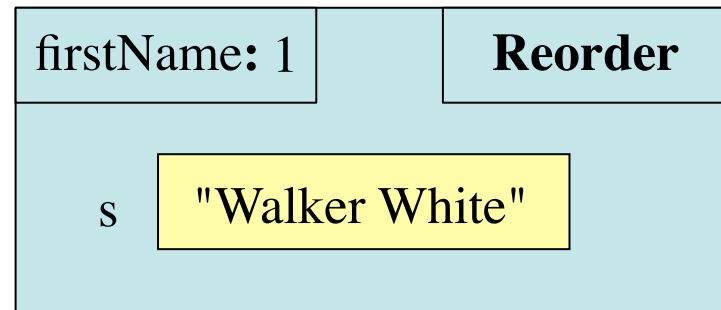
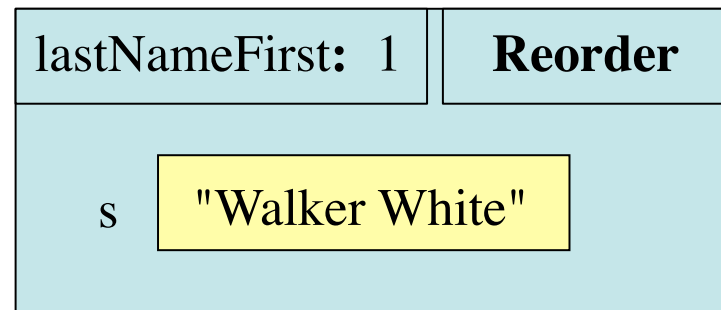
```
/**
 * Yields: first name in s
 * Precondition: s is in the form
 * <first-name> <last-name>
 * with one blank between names
 */
public static String
firstName(String s) {
    int end = s.indexOf(" ")
    return s.substring(0,end);
}
```

Frames and Helper Methods

Call: Reorder.lastNameFirst("Walker White")

```
public static String
lastNameFirst(String s) {
    String first = firstName(s);
    // Find the last name
    // Put together with comma
    return first; // Stub
}
```

```
public static String
firstName(String s) {
    int end = s.indexOf(" ");
    return s.substring(0,end);
}
```



Last Time: Stepwise Refinement

Exercise: Anglicizing an Integer

- `anglicize(1)` is “one”
- `anglicize(15)` is “fifteen”
- `anglicize(123)` is “one hundred twenty three”
- `anglicize(10570)` is “ten thousand five hundred

`/** Yields: the anglicization of n.`

`* Precondition: $0 < n < 1,000,000$ */`

`public static String anglicize(int n) {`

`// ???`

`}`

String: Revisited

- String is an unusual object
 - Do not create with `new`
 - Does not have named fields (that we know of)
- Data arranged in a “list”
 - List of characters
 - Access characters by position, not field name
 - **Method:** `charAt(int)`
 - Position starts at 0

- `String s = "abc d";`

0	1	2	3	4
a	b	c		d

- `String s = "one\n\two";`

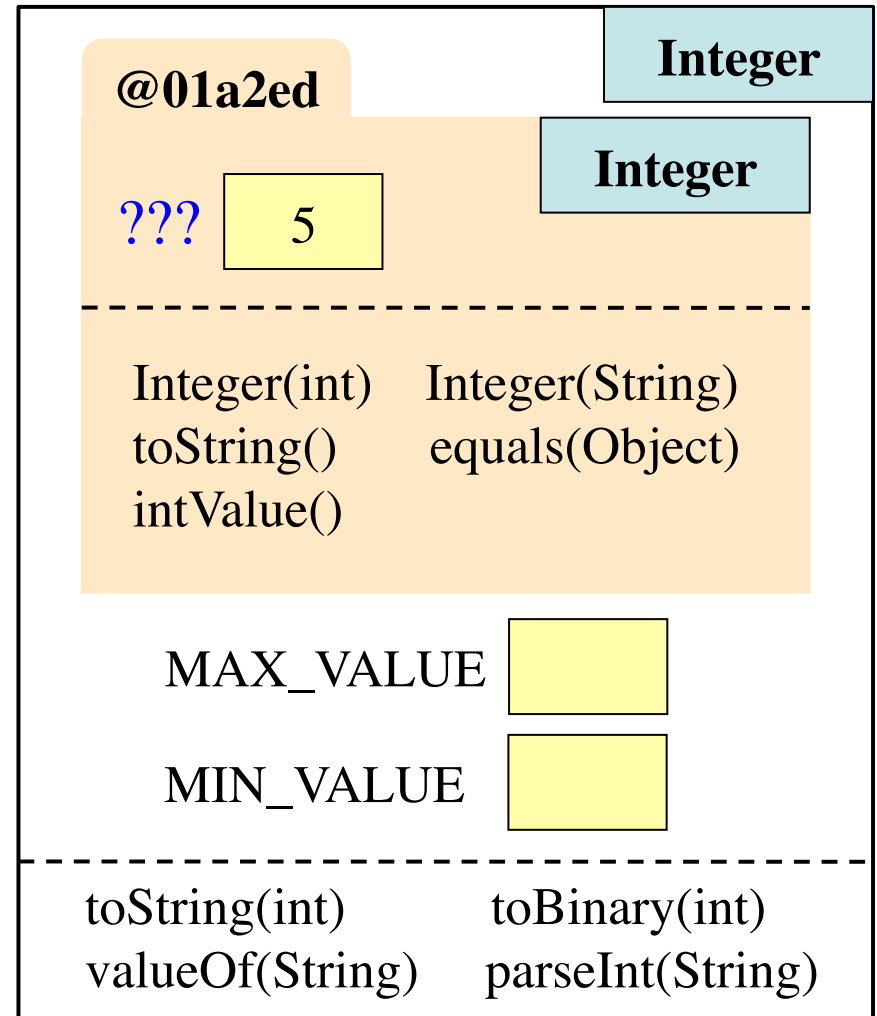
0	1	2	3	4	5	6
o	n	e	\n	t	w	o

Containers

- **Container**: an object that holds a list of objects
 - **But cannot hold primitive values (e.g. int, double, etc.)!**
- Java has several container classes
 - They are all in package `java.util`
 - **Generic classes**: type depends on what is contained
 - Put contained type in `< >`
- **Example**: Vector
 - `Vector<String>`: Vector that holds String objects
 - `Vector<AcornProfile>`: Holds AcornProfile objects
 - `Vector<Vector<String>>`: ????
 - `Vector<int>`: **NOT ALLOWED!**

Wrappers: Turn Primitives into Objects

- Want `Vector<int>`
 - `int` is primitive type, not class
 - Need to convert an `int` value (e.g. 9) into an object
- `Integer`: a **wrapper class**
 - Contains or wraps one value
 - Value cannot be changed: it is *immutable*
- Many useful static features
 - `Integer.MAX_VALUE`
 - `Integer.parseInt(String)`



Each Primitive Type Has a Wrapper

- When you need to treat a primitive value as an object, then just wrap the value in an object of the wrapper class.

Primitive Type	Wrapper Class
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Each wrapper class has:

- Instance methods
(e.g. equals, constructors, toString)
- Static variables and methods
(for useful computations)

```
Integer k= new Integer(63);
```

```
int j= k.intValue();
```

You don't have to memorize the methods of the wrapper classes. But be aware of them. See Section 5.1 and PLive 5-1 and 5-2 for more.

Boxing and Unboxing

- Modern (post 1.4) Java boxes/unboxes
- **Boxing**: Automatically add a wrapper
 - `Integer s = 4;`
 - Same as `Integer s = new Integer(4);`
- **Unboxing**: Automatically remove a wrapper
 - `int x = new Integer(4);`
 - Same as `int x = new Integer(4).intValue();`
- Type is determined by the variable assigned

Boxing and Unboxing

- Modern (post 1.4) Java boxes/unboxes
- **Boxing:** Automatically add a wrapper
 - `Integer s = 4;`
 - Same as `Integer s = new Integer(4);`
- **Unboxing:** Automatically remove a wrapper
 - `int x = Integer(4);`
 - `int x = new Integer(4).intValue();`
- Type is determined by the variable assigned

Avoid doing this; can be confusing

Example: Vector

- Create an empty vector instance (of Strings)
`import java.util.Vector;`
`Vector vec = new Vector<Integer>();`
- Add some strings to it
`vec.add(new Integer(2)); // Adds 2 at position 0`
`vec.add(new Integer(7)); // Adds 7 at position 1`
`vec.add(new Integer(-3)); // Adds -3 at position 2`
- Get the String at position 1
`vec.get(1) // Function call, gives 7`
- Search vector for number 5
`vec.indexOf(new Integer(5)) // Not found; gives -1`

Vectors Can Add and *Remove*

- Do the following:

```
import java.util;
```

```
Vector vec = new  
    Vector<String>();
```

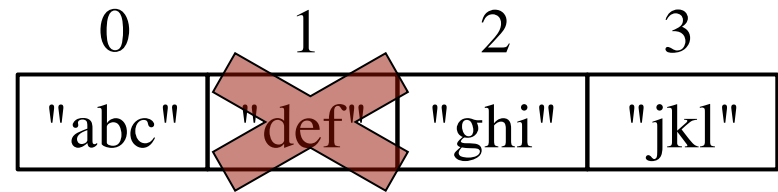
```
vec.add("abc");
```

```
vec.add("def");
```

```
vec.add("ghi");
```

```
vec.add("jkl");
```

```
vec.remove(1);
```



- After all this, what is the value of `vec.get(2)`?

A: Function gives "abc"

B: Function gives "ghi"

C: Function gives "jkl"

D: I have no clue