

CSE 11: Lecture 4

- ✓ Comments in code and coding style
- ✓ Procedural and object-oriented programming
- ✓ Designing classes for object-oriented programming
- ✓ Abstract Data Types (ADT's)
- ✓ Instance variables and instance methods
- ✓ Variable and method visibility
- ✓ Constructors
- ✓ The “this” variable
- ✓ Variables of reference type vs. variables of primitive type

(Reading: Savitch, part of Ch. 2, Ch. 4, and Ch. 5)

Comments

- ✓ Comments are things written in your program to make it more understandable to a human who has to debug, modify, or maintain your code

Note: This human might be you! Especially, you at a later time when you've forgotten half of what you knew when you were writing the code in the first place.

- ✓ Comments are ignored by the compiler; they are for human consumption
- ✓ Java has 3 styles of comments:
 - ✗ “double-slash” comments: these start with `//` and go to the end of the line
 - good for comments at the end of a statement, or one-line comments
 - ✗ “C-style” comments: these start with `/*` and go to the next `*/`
 - good for multiline, blocks of comments
 - ✗ “Javadoc” comments: these start with `/**` and go to the next `*/`
 - these are used by the javadoc utility to automatically generate HTML documentation of your code: more about this later in the course

Commenting, cont'd

- ✓ Q: When should you use comments?
- ✓ A: When they improve the understandability of your code
- ✓ Do not over-comment; do not just re-state the obvious:
 - ✗ this is a lousy comment:

```
double size;        // the variable that holds the size
```
 - ✗ this is a better comment:

```
double size;        // of the object, in pixels
```
- ✓ It's good practice to put a comment block at the top of each source code file that specifies the purpose of the program, the author, the date, and similar information.
- ✓ It's also good practice to put a comment block before the start of each public class, public method, and public variable definition to document their use
- ✓ And it's good practice to comment statements within method definitions to clarify what's going on there
- ✓ But clearly written code, with good choice of variable, class, and method names, and properly indented, does not need a lot of comments

Coding style

- ✓ The Java compiler doesn't care about indentation or spacing... your entire program can be written on one (very long) line and it will still compile
- ✓ But indentation that accurately reflects the structure of your code can make a big difference in the readability of your programs for humans
- ✓ Indent 2 or 4 spaces per level (1 space is too little, 8 spaces is too much)... example:

```
public class Foo {  
    public static void main(String args[]) {  
        while (true) {  
            System.out.println("I'm stuck in a loop!");  
        }  
    }  
}
```

- ✓ Keep each line of code short... less than 80 characters, so it fits in a standard terminal window
- ✓ The overall goal is: *understandability*, which makes code easier to write, test, debug, and use
- ✓ Examples and guidelines for all this can be found in Ch 2 of the textbook

Static and instance methods

- ✓ There are two kinds of methods that can appear inside a Java class:
 - ✗ static methods
 - ✗ instance methods
- ✓ If you use the keyword **static** in the method header (just before the return type), it is a static method; if you do not use that keyword, it is an instance method
- ✓ A static method is associated with the *class* within which it is defined
 - ✗ You do not need to create an object that is an instance of the class to use (invoke or call) a static method
 - ✗ Static methods are also called “class methods”
- ✓ An instance method is associated with *each object* that is an instance of the class within which it is defined
 - ✗ You (or someone) must create an object that is an instance of the class to use (invoke or call) an instance method
- ✓ We will now start getting into real Object-Oriented Programming, involving classes, objects, instance methods, and instance variables

Procedural Programming and Object-Oriented Programming

- ✓ “PP” and “OOP” are two programming paradigms, but they have much in common
- ✓ It is possible (though somewhat inconvenient) to do object-oriented programming (OOP) in a “procedural” language like C or Pascal
- ✓ It is possible to do procedural programming in an “object-oriented” language like Java
- ✓ Some things these two paradigms have in common are:
 - ✗ they use variables (see lecture 2)
 - ✗ they use branching (see lecture 3)
 - ✗ they use iteration (see lecture 3)
 - ✗ they use functions (see lecture 3)
- ✓ So, when learning OOP, don’t forget what you already know about programming
- ✓ (However, OOP does involve some different ways of thinking, as we will see)

Objects in object-oriented programming (OOP)

- ✓ Objects are a powerful way to think about computer programming
- ✓ An OOP object is something like a real-world object, and this is supposed to make it natural for programmers to think OO-style:
 - ✗ 1. Every object has some properties
(a pencil can be sharp or dull, a pen has a certain color it writes in, etc.)
 - ✗ 2. Every object has some possible behaviors
(a pencil can write or erase, you can take the cap off a pen and write with it, etc.)
 - ✗ 3. Every object is an instance of some class
(each existing pencil is a pencil --- it is also a writing implement, a physical object, etc., etc.)
- ✓ OOP objects are different from real-world objects though... They are software, and exist as bits in computer memory

Two important things about an object

- ✓ 1: Data the object contains (these represent the object's state or properties)
 - ✗ called *instance variables* in Java
 - ✗ (also sometimes called “member data” or “fields”)

- ✓ 2: Functions the object has (these carry out the object's behavior, or what the object does)
 - ✗ called *instance methods* in Java
 - ✗ (also sometimes called “member functions”)

- ✓ So you can think of an object as a thing that encapsulates...
 - ✗ some data, together with...
 - ✗ some operations that can be performed on that data

Classes in object-oriented programming

✓ A third important thing about objects:

3. Every object is created as an *instance* of a *class*

✓ In Java, a class is a datatype, but not a primitive type: Java classes are reference types

✓ Where does an object get the kinds of properties and behavior that it has?

✗ *The definition of the object's class* specifies what kind of properties and behavior the object will have

- properties <--> instance variables
- behavior <--> instance methods

✗ (Note that a class may define static variables and static methods as well. But these belong to the class as a whole; they are not properties and behaviors of each object separately.)

✓ So, in Object Oriented Design, when designing a class, you need to decide what properties and behavior you want objects that are instances of that class to have

An example of objects: Strings in Java

✓ To review:

- ✗ An instance of a class is called an *object*
- ✗ An object contains some data (instance variables), and provides some operations (instance methods) that can be performed on that data
- ✗ The type of data an object contains, and the methods it provides, are determined by the definition of the class the object is an instance of

✓ **String** is a *class* in Java (it is not a primitive type):

- ✗ Instances of the **String** class are objects
- ✗ The data contained by a **String** object is a sequence of **chars**, and the instance methods provided by a **String** object let you do things with its sequence of characters
- ✗ These things are determined by the definition of the **String** class in the **java.lang** package in the Java standard library
- ✗ (In addition (is this a pun?) to **String** instance methods, the Java language provides the **+** operator to perform concatenation on **Strings**)

Some useful String instance methods

- ✓ To determine the length of a String object, use its `length()` method, which returns an int:

```
String s = "Hi There!";  
System.out.print(s.length()); // prints 9
```

- ✓ chars in a String object are indexed in order by integers 0,1,..., up to the length of the string. To find out the char at a given position in a String, use its `charAt()` method:

```
String s = "Hi There!";  
char c = s.charAt(1);  
System.out.print(c);           // prints i
```

- ✓ To compare two String objects, you can use the `compareTo()` method of one of them, and pass the other as argument. `s1.compareTo(s2)` returns 0 if the `s1` is the same as `s2`, a negative number if `s1` comes before `s2` in lexicographic (same as alphabetic, if the Strings are both uppercase or lowercase) ordering, and a positive number otherwise

```
String str1 = "aardvark"; String str2 = "armadillo";  
int comp = str1.compareTo(str2);  
/* is comp 0, positive, or negative? */
```

- ✓ A String object has about 30 more instance methods, some of which are described in the textbook

User-defined classes

- ✓ You already basically know how to define a class in Java
- ✓ If you want to create objects that are instances of a class, the class should define some instance variables and instance methods, which determine the properties and behaviors of the objects
- ✓ How to do that? Easy:
 - ✗ Any variable declared outside of a method body, if it is not declared static, is an instance variable
 - ✗ Any method, if is not declared static, is an instance method
 - ✗ (You usually also want to define *constructors*, which are done somewhat differently from other methods. Details in a moment)
- ✓ The hard part is designing the class: determining what instance variables and instance methods you want its objects to have

Abstract Data Type design and implementation

- ✓ A *data type* is defined as: A collection of possible values, together with a set of operations on those values
- ✓ Example: the `int` data type in Java consists of the set of 32-bit signed integer values, together with such operations as `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`, etc.
- ✓ An *Abstract Data Type* (ADT) is a data type that is designed using abstraction: the principles of information-hiding
 - ✗ In an ADT, the implementations of the data values, and the operations on the data values, are designed as “black boxes” that you do not need to see inside in order to use
 - ✗ So, the ADT can be used without having to worry about the details of the implementation
- ✓ In an OO language such as Java, it is natural to implement a user-defined ADT by defining a class...
... so issues of ADT design become issues of class design

State- and behavior-hiding in OO ADT design

- ✓ Some methods of an object should be accessible from “outside the object”
 - ✗ these are *public* methods: they are accessible wherever the object itself is accessible
 - ✗ public instance methods represent the possible behaviors of the object that can be initiated from outside the object
 - ✗ they are part of the *user interface* to the object
- ✓ Normally, instance variables (and perhaps some methods) of an object should *not* be directly accessible from “outside the object”
 - ✗ these are *private* variables and methods:
they are accessible only from instance methods of the object itself
 - ✗ private instance variables implement the state of the object (inside the “black box”)
 - ✗ they are part of the *implementation* of the object
 - ✗ (use public instance methods to provide an interface to access the private variables if needed)
- ✓ Good ADT design, with limited access to object state and behavior, helps control bugs and makes both using and maintaining the software easier

Visibility of variables and methods

- ✓ Visibility of a member (variable or method) is determined by the visibility specifier keyword used in the definition of the member
- ✓ Visibility for instance methods and variables is the same as for static methods and variables...
 - ✗ The difference is: to refer to an instance variable or method at all, there must be an object whose instance variables and methods you will access
- ✓ Possible kinds of visibility are:
 - ✗ **private** visibility: can be seen only from methods defined in the same class
 - ✗ *package* visibility (no visibility specifier): can be seen from methods defined in the same class, or in another class in the same package. (For now: classes in the same directory are in the same package)
 - ✗ **protected** visibility: can be seen from methods defined inside the same class, or in a subclass (more about this when we talk about inheritance), or in another class in the same package
 - ✗ **public** visibility: can be seen from methods defined in any class
- ✓ We'll concentrate on **private** and **public** visibility for now

Referring to instance variables and methods

- ✓ When you call an instance method or refer to an instance variable, it must be the instance method or variable of a particular object (that's what "instance" means)
 - ✗ ... Savitch calls this object the "calling object". Other OO traditions call this "the receiver", and call the instance method call with its arguments "the message"
- ✓ If you have a way of referring to an object, you can refer to its instance members -- if they're visible -- using the dot operator: `<object_reference> . <member_name>`
- ✓ Or, if you are within the body of an instance method of the calling object, you can refer to instance members of the calling object *just by using the member name*
- ✓ Or, if you are within the body of an instance method of the calling object, you can use the automatically defined `this` variable to refer to the calling object. So within an instance method you can refer to an instance member with an expression of the form
`this . <member_name>`
 - ✗ ... and so sometimes the calling object is called "the 'this' object"
- ✓ Note: A static method does not have a calling object, and there is no `this` variable automatically defined for static methods!

Constructors

- ✓ A constructor is sometimes considered an instance method, but it is a special kind
- ✓ A constructor for an object is called when the object is created: it is the job of the constructor to initialize the object's instance variables appropriately
- ✓ If you don't define a constructor:
 - ✗ Java will give you a “default” constructor, which takes no arguments, and initializes instance variables to default values: zero for numeric types, false for boolean type, null for reference types
- ✓ So usually it is a good idea to define your own constructors:
 - ✗ you can define constructor(s) for a class, to deal with initialization of instance variables and whatever else you want to do each time an instance of the class is created
 - ✗ writing constructor functions lets you ensure that your objects' member data are initialized the way you want
 - ✗ Note: if you define any constructors at all, Java will not provide a default one that takes no arguments; but instance variables you do not initialize are still always initialized to default values automatically

Defining constructors

- ✓ Constructors are defined as instance (not static!) methods
- ✓ A constructor *must* have the same name as the class
- ✓ A constructor has *no return type* (not even void!)
- ✓ A constructor may take arguments
- ✓ A constructor may be overloaded (i.e. you can define different versions that take different number and/or type of arguments)
- ✓ Constructors should usually have public visibility
- ✓ Constructors are never invoked using the “dot” operator; they are used in conjunction with the **new** operator when creating objects

An example: A user-defined Circle class

- ✓ Suppose you are writing a program that does some graphics
- ✓ In an OO approach, you would want some classes that correspond to types of graphics objects you'll be dealing with: for example, you might want a Circle class
- ✓ In designing your Circle class, ask and answer these questions:
 - ✗ What properties does a circle have? These should correspond to instance variables:
 - size, location; probably also color, fill pattern, ... etc.
 - ✗ What behaviors should a circle have? These should correspond to instance methods:
 - change its size, move to a new location, display itself, tell you its area, ... etc.
 - ✗ How should the properties of a circle be initialized when it is created?
 - This is the job of constructors

A possible definition of a simple Circle class

```
public class Circle
{
    // Instance variables
    private double radius;
    private int center_x;
    private int center_y;

    // Constructors

    // default ctor: initialize radius 1.0, location (0,0)
    public Circle() {
        center_x = center_y = 0;
        radius = 1.0;
    }

    // one argument ctor: initialize radius, location (0,0)
    public Circle(double radius) {
        this.radius = radius;
        center_x = center_y = 0;
    }
}
```

A definition of the Circle class, cont'd

```
// three argument ctor: initialize everything
public Circle(double r, int x, int y) {
    radius = r; center_x = x; center_y = y;
}
```

```
// "copy constructor": initialize from another Circle
public Circle(Circle other) {
    radius = other.radius;
    center_x = other.center_x; center_y = other.center_y;
}
```

A definition of the Circle class, cont'd

```
// Instance methods

// set the location of the center of the circle
public void moveCenter(int x, int y) {
    center_x = x; center_y = y;
}

// set the radius of the circle
public void setRadius(double radius) {
    this.radius = radius;
}

// return the area of the circle
public double areaOf() {
    return Math.PI * radius * radius;
}

// display the circle at its current location
public void draw() { /* some GUI code, left out for now */ }
}
```

A use of the “this” variable

- ✓ Consider the setRadius instance method of the Circle class. The radius instance variable is defined as

```
private double radius;
```

- ✓ Now this version of the setRadius method would work:

```
public void setRadius(double r) {  
    radius = r;  
}
```

- ✓ But this does not (the formal parameter hides the instance variable):

```
public void setRadius(double radius) {  
    radius = radius; // doesn't do anything useful!  
}
```

- ✓ If you want, you can use the “this” variable to explicitly refer to the calling object, and access its instance variable:

```
public void setRadius(double radius) {  
    this.radius = radius;  
}
```

Calling constructors

- ✓ Unlike other methods, constructors are *never* called using the dot operator (reason: they are called to initialize an object which doesn't fully exist yet!)
- ✓ Constructors are called when you create a new object, using the **new** operator:

```
Circle c = new Circle();    // c refers to a Circle with  
                           // default location and radius
```

```
Circle c2 = new Circle(3.0); // c2 refers to a Circle with  
                           // default location and radius 3.0
```

```
Circle c3;
```

```
c3 = new Circle(10.0, 200, 300); // c3 refers to a Circle with  
                               // location 200,300 and radius 10.0
```

```
Circle c4 = new Circle(c3);    // c4 refers to a "copy" of c3
```


Using the Circle class

- ✓ Create circle objects, and call their methods to get them to do things!

```
public static void main(String args[]) {  
    int locx, locy, rad;  
    Circle circ = new Circle(); // create a new Circle object  
  
    System.out.print("Enter circle x,y location and radius: ");  
    locx = SavitchIn.readInt(); locy = SavitchIn.readInt();  
    rad = SavitchIn.readInt();  
  
    // move the circle there... call one of its methods  
    circ.moveCenter(locx, locy);  
    // make it that size... call one of its methods  
    circ.setRadius(rad);  
    // make it appear... call one of its methods  
    circ.draw();  
    // compute its area... call one of its methods  
    System.out.println("The circle has area " + circ.areaOf());  
}
```

A difference between Java primitive types and reference types

- ✓ In many contexts, Java primitive types and reference types behave differently. Here is an example:

- ✓ A declaration of a primitive type variable actually creates a variable of that type:

`double d; // this creates a 64-bit floating point variable`

- ✓ But a declaration of a reference type variable in Java does NOT create an object that is an instance of that type!

`Circle c; // this does NOT create a Circle object!`

- ✓ A declaration of a reference type variable by itself only creates a *reference* (or *pointer* or *handle*) that can be used as a way to refer to an object that is an instance of that type

- ✓ To actually create an instance of a class (an object) in Java, you use the `new` operator:

`c = new Circle(); // creates a Circle, stores reference in c`

- ✗ (Strings are an exception: you can use a string literal constant to create a String object, without using `new`, if you want)

References, pointers, and addresses

- ✓ A variable of reference (i.e. nonprimitive) type acts like a reference or pointer to an object of that type... how does it do that?
- ✓ As you know, a variable is just a region of computer memory that contains a pattern of bits that is interpreted in some way, depending on the type of the variable
- ✓ Also, you know that these regions of computer memory have addresses which specify their location in memory
- ✓ In Java, a variable of reference type is a 32-bit region of computer memory which contains a pattern of bits which is interpreted as the address of another region of computer memory!
- ✓ This address should be the address of an object of that reference type... this is what makes the reference (or pointer) refer (or point) to that object

More differences between Java primitive and reference types

- ✓ In Java, references (pointers) can never point to primitive type variables
 - ✗ (though of course, a reference can point to an object that contains primitive type instance variables)
- ✓ There are only two operations permitted on Java references (pointers):
 - ✗ They can be dereferenced:
 - When used with the dot operator (or array index operator), the address contained in the pointer is used to access the instance methods and variables of the object pointed to
 - ✗ Their values can be copied:
 - When used with the assignment operator or when passed as an argument to a method, the address itself is copied from one reference (pointer) variable to another
 - ✗ (Languages such as C and C++ permit many other operations on pointers; but Java's two operations are powerful enough for almost all applications, while helping to keep software systems simple and safe.)
- ✓ This situation is best visualized in pictures, so let's look at an example...

Creating instances of a primitive type: frame 1

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

*creates two variables of type int
(not yet initialized)*

```
int b,c;
```

b:

c:

Creating instances of a primitive type: frame 2

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

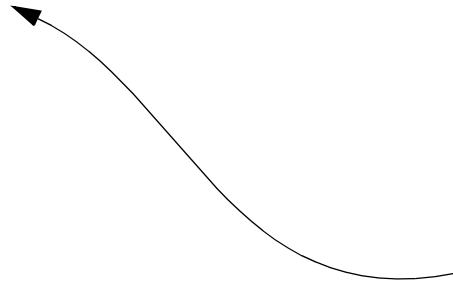
```
int b,c;
```

b: 3

c:

```
b = 3;
```

set value of b



Creating instances of a primitive type: frame 3

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

```
int b,c;
```

b: 3

c: 3

```
b = 3;
```

```
c = b;
```

assign value of b to c

(b and c are still different variables!)

Creating instances of a primitive type: frame 4

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

```
int b,c;
```

b: 3

c: 99

```
b = 3;
```

```
c = b;
```

```
c = 99;
```

(changing the value of c does not change b)



Creating instances of a class type: frame 1

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

*create two pointers to Circle objects.
(They don't point anywhere yet)*

```
Circle b,c;
```

b:

c:

Creating instances of a class type: frame 2

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

*create new Circle object, make b point to it.
The constructor initializes the instance variables*

```
Circle b,c;
```

```
b = new Circle();
```

b:

A diagram showing a variable 'b' pointing to a new object. A blue arrow originates from the 'b' label and points to the top-left corner of a large rectangular box representing the object. This box contains three fields: 'radius' with value '1.0', 'center_x' with value '0', and 'center_y' with value '0'. Above the 'b' label is a smaller, empty rectangular box representing the memory location of the variable.

c:

A diagram showing a variable 'c' with an empty rectangular box next to it, representing its memory location. No arrow points to it, indicating it has not yet been assigned an object.

radius:

1.0

center_x:

0

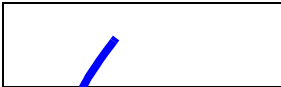
center_y:

0

Creating instances of a class type: frame 3

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

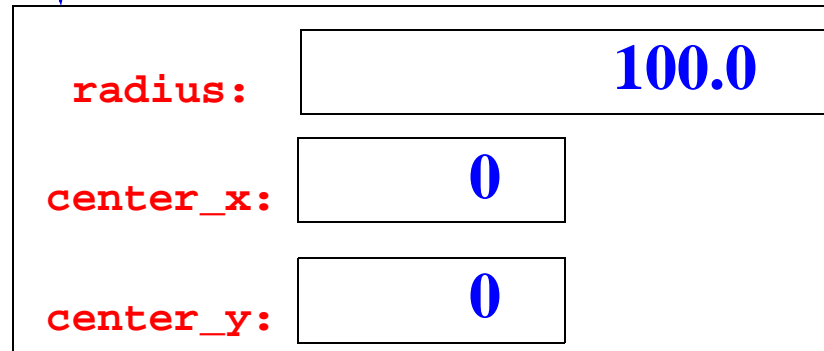
```
Circle b,c;
```

b: 

c: 

```
b = new Circle();
```

```
b.setRadius(100.0);
```



*dereference b to set radius
of the object b points to*

Creating instances of a class type: frame 4

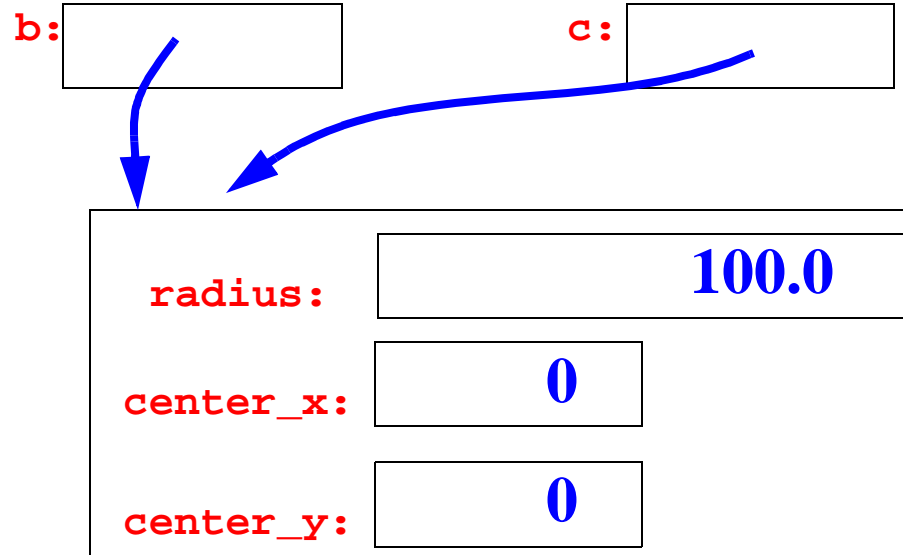
```
public static void main (String args[]) {
```

```
Circle b,c;
```

```
b = new Circle();
```

```
b.setRadius(100.0);
```

```
c = b;
```



copy address in b to c

(two pointers, pointing to one object!)

Creating instances of a class type: frame 5

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

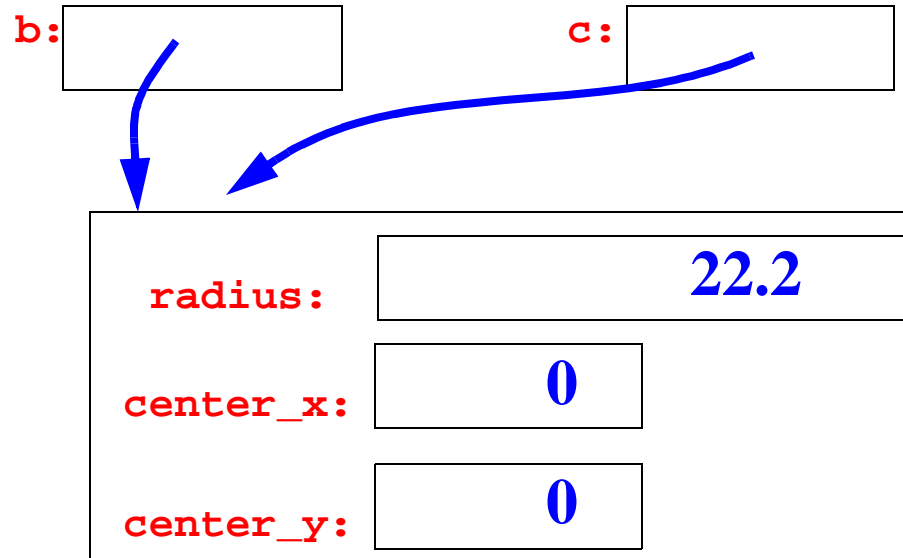
```
Circle b,c;
```

```
b = new Circle();
```

```
b.setRadius(100.0);
```

```
c = b;
```

```
c.setRadius(22.2);
```



*changing property of what c points to
does change property of what b
points to!*

Primitive types, class types, assignment, and arguments

- ✓ As shown in those examples, assignment has a somewhat different semantics for primitive types vs. objects:
 - ✗ if **a, b** are variables of primitive type, **a=b** copies **b**'s value into **a**'s storage location, which makes **b** and **a** have the same value (though they remain separate variables)
 - ✗ if **a, b** are variables of reference type, **a=b** copies the address stored in **b** to **a**'s storage location, which makes **b** and **a** point to the same object (though they remain separate pointer variables)
- ✓ Recall that in Java, when a method is called, values of actual arguments are in effect *assigned* to the corresponding formal parameters in the method definition
- ✓ This will mean that primitive types and objects behave differently when passed as arguments:
 - ✗ primitive type arguments are passed by value: the called method's formal parameter cannot be used to access the actual parameter (only a copy of its value)
 - ✗ object arguments are passed by reference: the called method's formal parameter CAN be used to access the actual parameter... they are both pointers to the same object!

Next time:

- ✓ Variable visibility
- ✓ Arguments passed by value and by reference
- ✓ ADT's, visibility, and object-oriented design
- ✓ Equality testing with `==` and `equals()`
- ✓ Java wrapper classes
- ✓ Methods for parsing text
- ✓ Mixing data types in numerical expressions and assignment
- ✓ Casts

(Reading: Savitch, parts of Ch. 4 and Ch. 5)