CSC142 Supplemental Reading: Week04

Contents

1 Object-Oriented Programming

Java is an object-oriented programming (OOP) language. This statement describes the programming paradigm, or style, supported by the Java language. OOP is just one of many different programming paradigms. Let's compare it to two other common styles.

1.1 Procedural Programming

The basic idea of procedural programming is this: a computer program can be viewed as a **set of instructions** that brings about some action. The procedural style is common in the real world. Imagine a recipe of Eggs Benedict. The recipe could be written as a sequential series of steps to be performed by the cook, resulting in the completed product.

To extend the cooking analogy, imagine the difference between a recipe written for a master chef compared with one written for a novice. A master chef operates at a higher level of **abstraction** -- she can understand instructions the novice might not. For example, one instruction in the master chef recipe might be "poach an egg for each serving." The novice, however, might need to look elsewhere in the cookbook for a separate procedure describing how to poach an egg. The instruction "poach an egg" is a higher-level abstraction. Just as the cookbook might have a separate procedure (recipe) to explain this, procedural programming languages allow the programmer to build new abstractions by writing new procedures. It's like adding new, more meaningful words to your vocabulary!

1.2 Functional Programming

We can think of computer programs as **executable math**. For example, consider the formula for the area of a circle:

area = PI * radius^2

We can use expressions like this in a computer program. We can also organize code into **functions**, so that the statement above might be changed to this:

area = circleArea(radius)

The function circleArea gets information into it through a **<u>parameter</u>** (located inside the parentheses) and **<u>returns</u> a result**, which is assigned to the variable Area. Most of your intuitions and knowledge from studying mathematics apply to computer programming.

1.3 Object-Oriented Programming

Object-oriented programs solve problems by **modeling** (simulating) things. An object-oriented program can be thought of as a group of objects (agents) that interact with each other by sending messages back and forth.

Consider a real-world example. Imagine that I want to send flowers to my mother for her birthday. I contact a local florist, choose from a range of available arrangements, and specify the delivery date. The florist tells me the cost. I offer a credit card for payment. The florist processes the payment through the credit card company. Because my mother lives more than 2000 miles from here, my local florist makes arrangements through an industry network for a second florist, located near my mother, to produce the arrangement and make the delivery. That florist works with its suppliers to obtain the flowers and other materials used in the arrangement. On the appointed day, the delivery is made and my mother is ecstatic!

Think about the agents involved in this routine transaction: me, my florist, the credit card company, the floral network, the remote florist, their suppliers, and my mother. Think about the information (messages) passed between the agents.

An object-oriented program works very much like this transaction. Each agent in the transaction would be a software object. Objects have qualities (or **properties** or attributes), and may be made up of other objects (parts). The complete set of all of an object's properties and their current values is called its **state**; if any property (or part) is changed, the object's state has been changed. Objects are also animated. They can respond to messages sent to them -- they exhibit **behavior**.

In the example above, my florist has, among other things, a telephone number, an inventory, and an employee who makes the floral arrangements. My florist also has the ability to process an order for flowers.

If we were modeling a car, we might say that the Car object is made up of, among other things, 4 Wheel objects, and its attributes might include *speed* and *direction*. The Car object may be able to respond to *accelerate* and *turn* messages with an appropriate change in state.

In this course, we will create software systems using Java that solve problems using OOP techniques. Although Java is fundamentally an objected-oriented programming language, you will find elements of all three of these programming styles in most programs.

1.4 Abstraction in Computer Programming

Imagine that you borrow a car from a friend -- a car you have never driven before. Would you be able to operate the car successfully? How do you know that the various parts of their car (ignition, steering wheel, accelerator, etc.) do the same thing as those parts of your car? The answer: you understand the **abstraction** of a *Car*. For example:

- A *Car* must have a steering wheel that causes it to turn right and left. Turning the steering wheel to the right must cause the *Car* to turn to the right.
- A *Car* must have an ignition switch that turns it on and off.
- etc.

1.4.1 Procedural Abstraction

How many of us have called the Math.sqrt() method? How many of you know the algorithm for finding the square root of a number? You've engaged in procedural abstraction: you've been able to use a procedure (method, function, whatever name you want to use) without having to know the details of the implementation.

Data Abstraction

How many of us have used the data type double? How many of us know how a double is stored in a computer? Again, abstraction is at work. We're able to use doubles without focusing on the bits inside the machine. We also understand that for different data, operators behave differently (think of division with ints vs doubles, or + with numbers vs Strings). Also, consider your work with the Scanner class, DrawingPanel class and others. As clients, you've been able to work with these objects, these defined data types, without knowing the details.

So we've had experience with data abstraction. It is important then, as we move to become the suppliers of new data types, as we define our own objects, to recognize that we are providing this abstraction for other client programmers. We want to provide functionality while hiding implementation at the same time (**encapsulation**).

1.4.2 Interface vs. Implementation

Even if the internal operation of the *Car*'s parts is different, the interface must be the same. Do you know how every part of a *Car* (steering, electronic ignition, anti-lock brake system, etc.) works? Probably not. But, you do know how to **use** a Car! You understand the **interface** of a *Car* even though you may not understand its **implementation**.

An OOP programmer would say that:

- your car is an **object**
- it is an **instance** of the *Car* **class**
- to use a *Car* object, you only need to know its **pubic interface**; there is no need to know anything about the **implementation** (and that will probably be hidden from you)

The public interface of an object is the set of behaviors it exposes to outside users. The implementation is the internal code that makes it work.

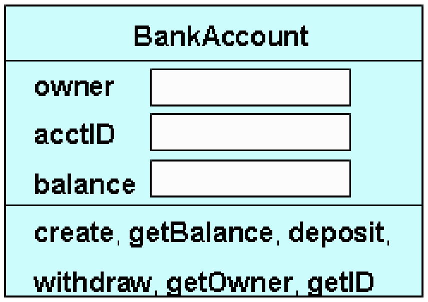
Object-oriented programming makes it easy to **reuse** code. There is no need to "reinvent the wheel" (no pun intended) when a new problem needs to be solved. Libraries of software classes exist that can be applied to solve problems. So, if you want to get a pizza delivery job, you do not start by designing a car. Instead, you go get a car that someone else designed, and then use it to do your job.

2 Creating and Using Objects

You will recall that we have said that an object is a construct that has properties ("state", or stored data) and functionality (behavior). In Java, a **class definition** is used to specify the properties and define the functionality for a given kind of object. In this note, we will examine the syntax for constructing and using objects using classes that already exist.

2.1 BankAccount Objects

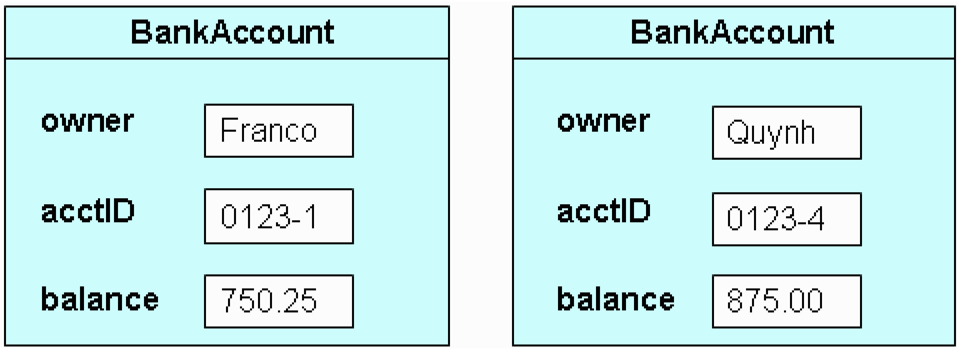
In this week's sample code folder, you will find a file named **BankAccount.java**. This file contains the definition for a class named BankAccount. A class is not the same thing as an object. In an object-oriented program, a **class** defines a new data type and serves as a template for creating objects



The figure at right contains a summary of the properties and functionality of a BankAccount. This is a model for all BankAccount objects that might be used in a program. There are 3 properties (owner, acctID and balance) and several operations. This grouping together of data and functionality (operations that can be performed on the data) is a key characteristic of software objects.

An object is a specific instance of the class to which it belongs.
Using a real world analogy, the word and concept "dog" is equivalent to a class definition. There are countless specific examples (instances) of a "dog" in the world -- each a little bit different. The same is true for software objects. Each **object** -- each **instance of a class** -- can be different. The

difference comes in the object's **state** -- the values of its properties. The figure below represents 2 different BankAccount objects.



Each BankAccount object has the same properties, but may have **different values** for those properties. Each BankAccount object is stored separately in memory, and the values of the properties are stored in **instance variables** (or **fields**). These values are initialized when the object is constructed, and can be changed later if desired.

Notice that a BankAccount object is stored as a **group of instance variables**. How is it possible to maintain access to an object? The answer is that we use a special kind of variable called a **reference variable** to hold (store) a reference to an object. We declare a reference variable by specifying its data type, and then choosing a name for it, like this:

```
BankAccount mySavings;
```

Notice that the name of the class is the data type used for the reference variable.

Every object must be created (or **constructed**, or **instantiated**) before it can be used. The following statement constructs a new BankAccount object and **binds** a reference to it to the variable mySavings:

```
mySavings = new BankAccount( "Maria", "123-55", 100.0 );
```

The functionality of an object is defined by code written inside the class definition. As you already know, a **method** is the basic unit of executable code in Java. Java allows two different kinds of methods: "static" or "class" methods -- the kind you have been writing up until now in this course, and "instance" methods (sometimes called "ordinary" methods, because the static methods are really the exception, even though we have covered them first). The functionality of an object is defined by writing instance methods. The basic syntax for using (calling) an ordinary method is:

```
objectName.methodName()
```

Notice that the name of the method is preceded by a 'dot' (the **member selection operator**) and the name of an object (the **qualifier**). Another way to think of a statement like this is sending a message to an object. For example, this statement:

```
mySavings.deposit( 8000.0 );
```

could be described as "**sending a *deposit* message** to the *mySavings* object" or "**calling the *deposit* method** of the *mySavings* object." The parentheses () are always required with a method call or constructor call, and may optionally contain one or more **arguments** (also called **actual**

parameters) if the method definition requires them. When a method call makes use of arguments, the arguments in the method call must match the parameters in the method definition in 3 ways:

- the **number** of arguments must match the number of parameters
- the **order** in which the arguments are listed must match the parameters
- the **data type** of each argument must "conform to" (think "match" for now) the corresponding parameter

Methods can generally be divided into two broad categories:

- A method that requests data from the object, but does not change its state, is called a **query method**. The BankAccount object has 3 of these: getOwner, getID and getBalance. A query method that directly returns the value of an instance variable, as all 3 of these do, may also be called an **accessor method**. All query methods **return** a value to the calling statement.
- A method that may **change the state** of the object is called an **update method** (or a **command**, or a **mutator method**, or simply a **mutator**). The BankAccount object has 2 of these: deposit and withdraw. Update methods may or may not return a value.

Here is an example of a query method being used. The variable myBalance has been declared to hold the value returned by the getBalance method:

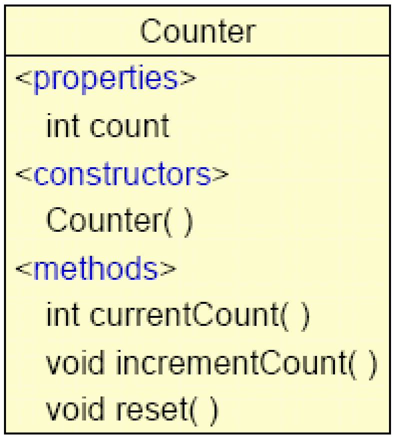
```
double myBalance = mySavings.getBalance();
```

Notice that the parentheses are required even though there are no arguments.

Java makes it easy for programmers to create excellent documentation by writing complete Javadoc comments in the code. Here is a link to the BankAccount class specification that was automatically generated from the comments in the BankAccount class. You can get a lot of information about how to **use** a class from its specification. You may notice that the properties are not shown in the class specification. This is because they were declared to be ***private***. The **class specification** is a view of the **public interface** of a class.

3 Class Implementation

Imagine that we wish to create a class that allows us to produce Counter objects. The specification for a Counter is provided in the figure to the right is what is referred to as a **class diagram** block. The class diagram block has a region at the top that shows only the name of the class. Below that is a region that lists the properties, constructors and methods of the class. Properties and methods are also referred to as members of the class. This block shows 5 items: 4 members and one constructor. We will talk more about class diagrams throughout the quarter.



3.1 Class Definition

Here is the basic structure for the definition of a class that is intended to serve as a template for creating objects:

```
public class ClassName {
   // instance variable declarations
   // constructor definitions
   // method definitions
}
```

The figure at right shows this model applied to the Counter class. In a Java class, each property of an object is implemented by declaring an **instance variable** (also called a **field**). An instance variable is a memory location where the value of an object's property can be stored. Since each object (each instance of a class) may have a different value for each property, a separate set of instance variables is allocated for each instance of a class. That is to say that **every object has its own, individual set of instance variables**. Instance variables are declared inside the class definition, but **outside** of any method or constructor definitions. By convention, instance variable definitions come before constructor and method definitions.

```
/**
 * A simple integer counter.
 */
public class Counter {

  private int count;

  /**
   * Create a new Counter, with
   * count initialized to 0.
   */
  public Counter ( ) {
    count = 0;
  }

  /**
   * The number of items counted.
   */
  public int currentCount ( ) {
    return count;
  }

  /**
   * Increment the count by 1.
   */
  public void incrementCount ( ) {
    count = count + 1;
  }

  /**
   * Reset the count to 0.
   */
  public void reset ( ) {
    count = 0;
  }

}
```

Instance variables are created when the object is constructed, and persist as long as the object that contains them exists. So, the **lifetime** of an instance variable is the same as that of its object. An instance variable may be used within any ordinary method in the class -- the **scope** of the variable is the entire class definition.

Recall that variables declared **inside** a method or constructor are referred to as **local variables**. The scope of a local variable begins on the line where it is declared and ends at the end of the **block of code** in which it is declared. A block of code is a series of statements surrounded by { } .

There are three method definitions in this class. Each one has the following structure:

```
public returnType methodName () {
     // method body
}
```

Notice that the keyword static is not used -- these are "ordinary" (or "instance") methods.

A **constructor** is similar in appearance to a method, but with two distinguishing features:

1. **A constructor definition never includes a return type -- not even the keyword *void*.** In fact, if a return type or the keyword *void* is accidentally added to the definition, then what is defined is not a constructor at all, but rather a method. This is an easy mistake to make, and is not a syntax error, so the compiler will not catch it -- be careful!
2. **The name of a constructor must match the name of the class exactly.**

The code in a constructor is executed when a new instance of the class (a new object) is created. The purpose of this code is to do whatever is necessary to initialize the object and get it ready to be used. At a minimum, **a constructor should initialize all instance variables**. Constructors may use formal parameters as needed. Just like with methods, the parentheses () following the name of each constructor are required even if no parameters are used.

Notice the use of the access modifiers *public* and *private*, and also the presence of complete Javadoc comments for each method and for the class as a whole.

The Counter class is an extremely simple example of a Java class. Look in this week's sample code for a more complete example of a class definition.

4 Reference Variables and Object Diagrams

By now you know that every variable in a Java program needs to be declared, and that a variable declaration must always specify a **data type**. Java is a strongly-typed language -- every variable has a data type, and a variable may store only information that conforms to that data type. The following two statements are variable declarations:

```
double amount;
BankAccount mySavings;
```

The first statement declares a variable named **amount** with the data type ***double***; the second declares a variable named **mySavings** with the data type **BankAccount**.

We have also seen already that there are two different categories of data types in Java: primitive types and reference types. Java has exactly 8 **primitive types**: ***byte***, ***short***, ***int***, ***long***, ***float***, ***double***, ***char*** and ***boolean***. Every variable of a primitive type stores a single piece of information, and that information is stored directly in the memory location associated with the variable.

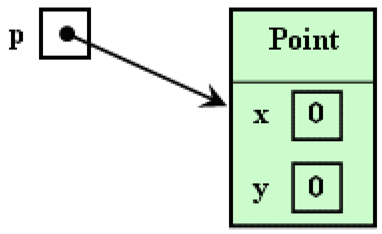
Every data type that is not a primitive type is a **reference type**. A variable declared with a reference type is called a **reference variable**: a variable that holds a reference to a software **object** (an **object reference**). What do we mean by a 'reference'? A reference is the **memory address** where an object is located. So, a reference variable holds an address, and at that memory address you will find the actual data stored in the object.

Consider the following statement:

```
Point p = new Point();
```

Let's look at what happens when this statement is executed. The **expression** new Point( ) constructs a Point object, and the **value** of the expression is the **memory address**[1] where that Point object is located. The statement also declared a new reference variable named p and stores in that variable the memory address of the new Point object. In OOP terminology, we say that **the statement binds the variable p to the Point object**. So, the statement **declares** a variable named p, **instantiates** a new Point object, and **binds** p to the Point object. A statement like this is said to follow the **declare-instantiate-bind pattern**.

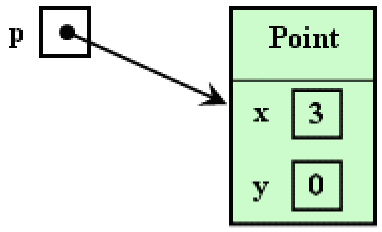
The reason object references are needed is that a single variable like p can hold only one piece of information, but an object is a data structure that can hold multiple pieces of information. So, when an object is instantiated, it gets as much memory as it needs, and then the address where that block of memory begins becomes the **reference** to that object.



Programmers use a special kind of diagram called an **object diagram** to describe the contents of memory and show how they change as a program is executed. The figure at right is an object diagram that shows the result after the statement above is executed. A small box in an object diagram represents a variable, and the name of the variable is shown adjacent to the box. A larger box represents an object -- and an object's instance variables are located inside the object's box. So, in this diagram, there are 3 variables. The variable p is declared in the statement above, and the variables x and y are instance variables of the Point object -- declared and initialized when the Point object is created. x and y are primitive variables; the value stored in a primitive variable is shown directly in the variable's box in an object diagram. p is a reference variable. The arrow represents the object reference stored in the variable p. **In an object diagram, an arrow (an object reference) always originates inside the box that holds the value of a reference variable and always points to an object.**

Once we have an object reference stored in a variable, we can use that reference variable to access the properties and methods of the object. For example, the following statement changes the state of the Point object:

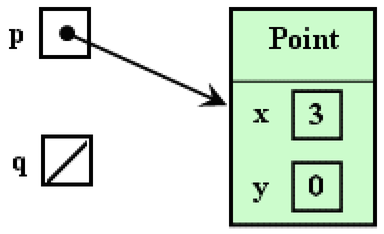
```
p.setX( 3 );
```



The object diagram at right shows the effect of this statement. Whenever you encounter a 'dot' (the member selection operator) in an expression, think of that as meaning 'follow the arrow' in the object diagram. That's how you determine which object executes the method.

The following statement declares a new reference variable:

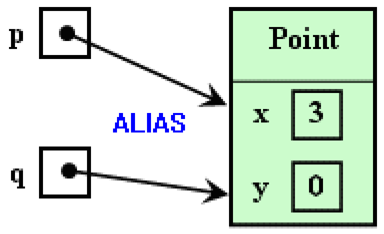
```
Point q;
```



No object has been bound to the variable yet. When a reference variable has no object bound to it, it has the special value ***null***. In an object diagram, we identify a ***null*** reference variable by putting a slash through its box. The diagram at right represents the state of memory after this statement has been executed.

This statement:

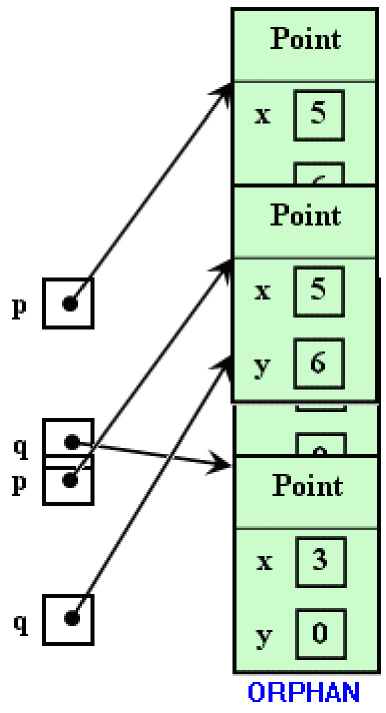
```
q = p;
```



changes the value of the variable **q**. Specifically, it causes **q** to have the same value as **p**. Since **p** holds a reference to the Point object, the result of this statement is that **q** holds a reference to the **same** object. When two variables hold references to the same object, we say that each of them is an **alias** for that object. Notice that, although the statement reads "**q = p**", the arrow from **q** does not point to the variable **p**. An arrow always points to an object. This makes sense if you remember that a reference variable actually holds a memory address. What the statement says is "store in the variable **q** the same memory address that is in the variable **p**." That means that **q** refers to the same object as **p**. (If I were reading this statement aloud, I would say "q **gets** p" because the equal sign is the assignment operator -- it assigns a value to a variable.)

When you have aliases, either variable may be used to access the object. For example, p.getX() and q.getX() would both return 3.

Look at this statement: `p = new Point( 5, 6 );`



What does it do? Does it declare a new variable? No. Does it construct a new object? Yes! (every time you see the keyword **new**, a new object is being created). Does it create a binding? Yes -- from **p** to the new Point object. The figure at right shows the object diagram after this statement has been executed. Notice that **p** now refers to a different Point object. There are now two Point objects in the diagram. Notice that each object has its own, independent set of instance variables. So, the expression **p.getX()** would evaluate to 5, but **q.getX()** would evaluate to 3. Remember to think of 'dot' as meaning "follow the arrow" -- that's how you will know which object to use, and that's why object diagrams are a valuable tool for analyzing programs.

We have one final statement to consider in this analysis. Let's execute this statement again:

```
q = p;
```

This statement has the same effect as before -- it causes **q** to refer to the same object as **p**. One consequence of this is that there is no longer any reference to the (3,0) Point. An object that has no references is called an **orphan**. Once an object is orphaned, it can never be used again. Java uses automated **garbage collection** to free up the memory associated with orphaned objects -- so eventually the orphan will simply go away and that memory will be available to use again. Garbage collection happens behind the scenes, and in most situations (and certainly in this course) you don't need to worry about it at all.

This week's practice exercise will give you some practice drawing object diagrams. This is a skill you need to learn in order to be able to analyze the programs you will write later this quarter. Be sure to read the note How to Draw an Object Diagram.

Both of the instance variables in the Point class are primitive variables. However, instance variables may also be reference variables. Formal parameters and local variables may also be reference variables.

4.1 References as Parameters

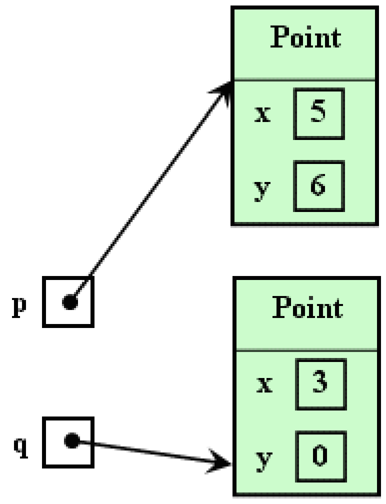
Let's examine two methods of the Point class in more detail to see how reference variables may be used as parameters and return values. The **midpoint** method is an example of a method that uses a parameter of a reference type. Here is its code:

```
public Point midPoint( Point other ) {
double midX = ( x + other.x ) / 2;
double midY = ( y + other.y ) / 2;
return new Point( midX, midY );
}
```

Notice that there is a formal parameter named **other**, and its data type is Point. When the **midpoint** method is called, the argument provided in the method call must be a reference to a Point object. The second line of code calculates the average value of the two x coordinates. How can there be two different values for x at the same time? Simple: every Point object has its own, separate instance variable named **x** -- we saw that in the object diagram analysis above. One of these instance variables belongs to "the object executing the midpoint method" and the other belongs to the object named '**other**'. To make this distinction clearer, the second line of code can be rewritten as follows:

```
double midX = ( this.x + other.x ) / 2;
```

The word ***this*** in the statement above is a Java **keyword** that represents **a reference to the object currently executing the code**. Another way to think of it is that the keyword ***this*** **allows an object to get a reference to itself**. The keyword ***this*** is implied as a qualifier when an instance variable or method is used from inside the class where it is defined. Showing it explicitly is generally optional[2], but some programmers like to do it consistently.



Imagine that we have been executing a program and have reached the state defined by the object diagram at right. There are two reference variables, and each is bound to a different Point object. Now, imagine that the following statement is executed:

```
Point mid = p.midPoint( q );
```

This statement sends a **midpoint** message to (calls the **midpoint** method of) the object referenced by **p**. The argument of the method call is **q**, another Point reference. So, when the code is executed, the statement above evaluates as follows:

```
double midX = ( this.x + other.x ) / 2;
4 <-- ( 5 + 3 ) / 2
```

Notice that **this.x** has the same value as **p.x** because the object **p** is executing the distance method.

This example also points out the fact that the **'dot' operator** (**member selection operator**) may be used to access an object's instance variables when they are visible ("within **scope**").

4.2 References as Return Values

The **midpoint** method also illustrates an object reference used as a return value. The expression **new Point(...)** evaluates to an object reference to a newly-constructed Point object. This reference is the value returned by the **midpoint** method. Notice that the **return type** of the method is specified to be **Point**, indicating that it should return a reference to a Point object. The **toString** method in class Point is another example of a method that has a reference type as its return type: type **String**.

Notes:

1. A memory address is a 32-bit **hexadecimal (base 16) number**. An example might look something like this if it were printed out: 016C3FD9. Memory for new objects is allocated in a special region of memory called the **heap**.

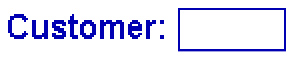
2. There is one important situation where the keyword ***this*** is required. If an instance variable (or class variable) and a parameter (or local variable) have the same name, then the keyword this needs to be used to achieve disambiguation. The identifier refers to the parameter by default, but the keyword ***this*** can be used as a qualifier to specify that you want to work with the instance variable. Here's an example:

```
public class Widget
{
   private double cost;
   ...
   public void setCost( double cost ) {
      this.cost = cost; // assign to the instance variable the value stored in the parameter
   }
   ...
}
```

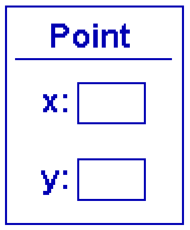
5 How to draw an Object Diagram

1. Use PENCIL. Bindings change!
2. For **each line of code**, ASK...
3.

- Does it **declare a variable**?
  - If so, draw it:
  - If no other binding is created, be sure to initialize newly-created variables. Numeric variables are initialized to 0, booleans to false, and all reference variables (including String variables) to null.



- Does it **instantiate an object**? Look for the **new**keyword. (See note 1)
  - If so, draw it, including placeholders for the instance variables:
  - When drawing a new object, **execute all the code in the constructor** and **ask all the same questions** at each line of code. Show **instance variables inside** the object, **other objects outside**.



- Does it **create a binding**? Look for the assignment operator (=).
  - If so, show it on the diagram.
    - Primitive types: fill in the value.
    - Strings: same as primitives (See note 3).
    - Reference types: show a reference link (arrow) from the variable to the object.

- If any **method calls** are present, **execute all the code in the method** and **ask all the same questions** at each line of code.

•

Notes:

1. A reference variable can only point to an object, not another reference variable.
2. A reference variable can only point to one object at time. If you make a new link, the previous one is lost.
3. A String is an object in Java. However, our diagrams will be clearer if we treat Strings in the same way we treat primitive values. Omitting the reference semantics from String references causes no errors in interpretation because String objects are immutable -- that is, once they are created, they cannot be changed. Thus, the third line in the code snippet below actually produces a new String object and binds it to s2...

This code:

```
String s1 = "Strong";
String s2 = s1;
s2 = s2 + "er";
System.out.println( s1 );
System.out.println( s2 );
```

Prints this in the terminal window:

```
Strong
Stronger
```

6 More on Drawing Object Diagrams

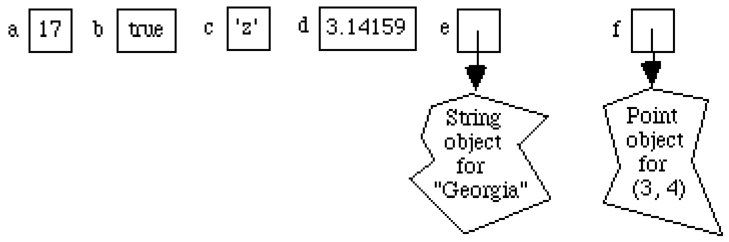
This document explains the rules for drawing object diagrams. (Adapted from a document used at Wellesley College)

6.1 Object Diagram Rules

Object diagrams consist of three kinds of entities: variables, objects, and classes. These are described below.

6.1.1 Variables

A variable is a named location that is depicted as a name next to a box. The box may contain a primitive data type element (i.e., an element of type **int**, **boolean**, **char**, **double**, and a few other types) or a reference to an object. For instance:



Variables are created in Java via one of the following two declaration statements:

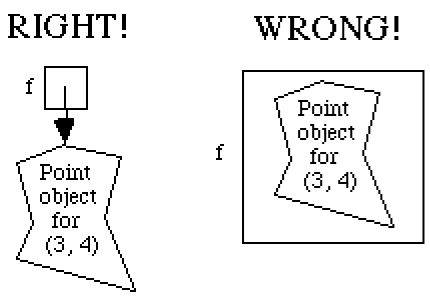
type name;
type name = *initial-value-expression*;

The first form creates a variable with a default value for the specified type. The second form creates a variable that holds the value of *initial-value-expression*, whose type must be *type*. For example, the above variables can be created by the following statements:

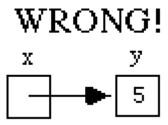
```
int a = 17;
boolean b = true;
char c = 'z';
double d = 3.14159;
String e = "Georgia";
Point f = new Point(3,4);
```

Important :

- A Java variable **never** has other boxes within it. It may contain a pointer to an object, but **never** contain the object itself.



- An object diagram **never** contains a pointer to a variable. For instance, the following picture is nonsensical in Java:



There are four kinds of variables:

1. **Local variables** are those declared in method bodies; these appear in the variable portion of a Java execution frame. Unless explicitly requested, you generally need not show execution frames in an object diagram.
2. **Instance variables** are non-static variables declared within a class body; these appear within an instance object box; see the discussion of objects below.
3. **Class variables** are static variables declared within a class body; these appear within a class box; see the discussion of classes below;
4. **Indexed variables** are numbered variables that appear within array objects; see the discussion of array objects below.

6.1.2 Objects

A Java object is a collection of variables. It is depicted as a labeled box containing its component variables. The label indicates which kinds of object it is. There are two kinds of objects: instance objects and array objects.

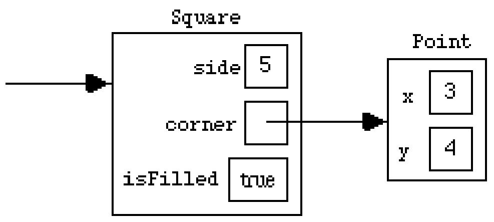
6.1.2.1 Instance Objects

An instance object is an object constructed from the template of a particular class. The variables within an instance object are the instance variables of the class. The label of an instance object is the name of the class from which the instance was derived.

Suppose the Square class has the following instance variables:

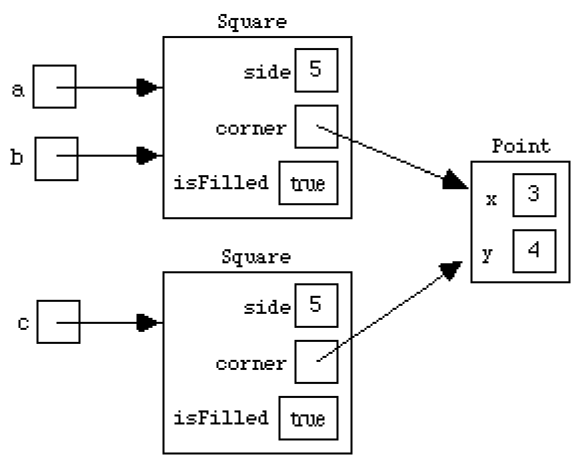
- An integer side specifying the side length of the square.
- A Point object corner specifying the upper left-hand corner of the square in a two dimensional grid.
- A boolean isFilled specifying whether the square should be drawn filled or not.

Here is the picture for an instance of a Square class:



Java programs never manipulate the instance box itself but only references (pointers) to the instance box. So technically we should say the "the corner variable contains a reference to a Point object". However, it is common to identify the reference to the object with the object itself, so we will often use the looser language "the corner variable contains a Point object". But it is important to keep in mind that a variable box can never contain sub-boxes in Java; it can only contain a reference to a box that contains variable boxes.

Because objects can never be manipulated without a reference to them, we will always draw objects as boxes with at least one reference to them, even though that reference may not be contained by any variable. The above diagram shows such a "free-floating" reference to the instance of the Square class.



In an object diagram, there may be many references to a particular instance box; all such references are considered to refer to the exact same object. For example, in the diagram at right, variables a and b contain the same Square object while c contains a different Square object. Both Square objects contain the same corner Point object.

The fact that object diagrams can show that the same object is "shared" by several variables is critical to understanding computation in Java. Sharing can be observed when an object is mutated -- that is, when the values of its instance variables change. For example, if the side of the Square contained in a is changed to 6, then the Square in b is also changed but the Square in c is not. If the x coordinate of the Point in the above diagram is changed to 7, then the Squares in a, b, and c all can "see" this change.

Even without mutation, sharing can be detected in Java by the equality operator ==. When == is applied to two objects references, it is only true if they are references to the same object. For example, in the above diagram, (a == b) is true but (a == c) and (b == c) are false. If the corner of a square can be extracted by a getCorner() method, then all of the following are true:

```
(a.getCorner() == b.getCorner())
(a.getCorner() == c.getCorner())
(b.getCorner() == c.getCorner())
```

The instance variables of a class are specified within a class declaration. Here is a skeleton of the class declaration for the `Square` class:

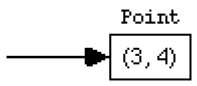
```
public class Square
{
   // Instance variables;
   private int side;
   private Point corner;
   private boolean isFilled;

   // Constructor
   public Square (i, p, b) {
      side = i;
      corner = p;
      isFilled = b;
   }
   // Instance Methods go here. ...
}
```

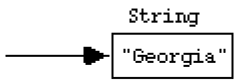
The **private** keywords in the instance variable declaration indicate that the variables can only be manipulated by the methods of the class; they cannot be directly extracted from the object.

Important: The only way to know what instance variables should be in an instance object is to examine the definition of the class of that instance.

Sometimes the class of an object is not available for inspection, in which case the private instance variables of the class cannot be determined. In this case, we must resort to a more abstract representation of the instance object. For instance, if we do not know that the Point class has instance variables named x and y, we can represent a Point instance with x-coordinate 3 and y-coordinate 4 as follows:



This sort of abstract representation is commonly used for instances of the String class, whose implementation details are not available for inspection:



Instances of a class are created by using new in conjunction with a constructor method for the class. Here is a sequence of statements that creates the objects shown in the two-square diagram above:

```
Square a = new Square(5, new Point(3, 4), true);
Square b = a;
Square c = new Square(5, a.getCorner(), true);
```

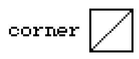
Note that the number of objects in the diagram (3) is exactly the number of **new** expressions that were executed. Since objects can only be created by **new**, this is always the case. (However, be careful: some uses of **new** are hidden. For instance, creating the `String` object `"Georgia"` involves a hidden use of **new**.)

Note that the corner of Square c is extracted from a via a.getCorner() (which is equivalent to b.getCorner()) -- this establishes the correct sharing shown by the diagram. It would be incorrect to write

```
Square c = new Square(5, new Point(3,4), true);
```

because this would create a second `Point` object, and there is only one in the diagram.

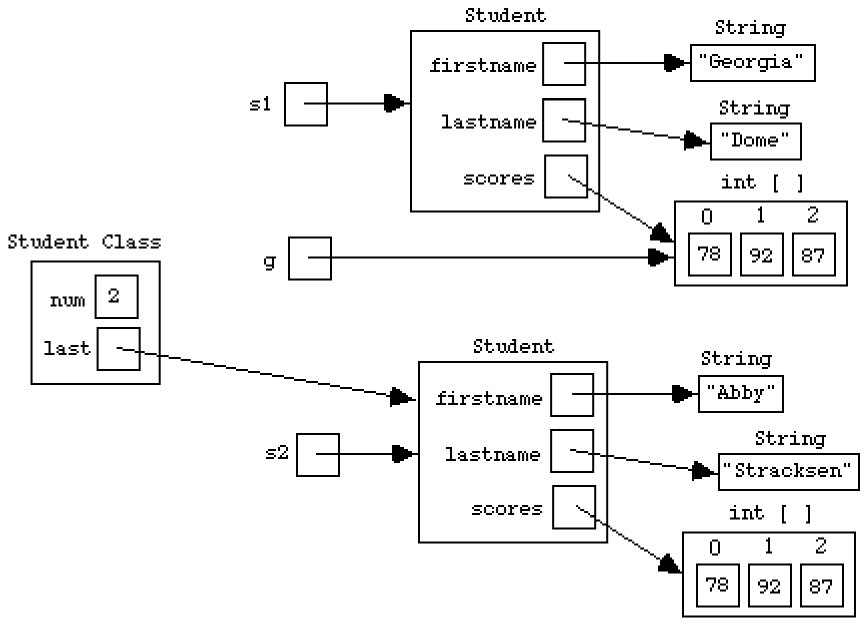
In many cases the object reference that belongs in an variable is not known when the variable is created. In this case, a special so-called **null reference** (written in Java as **null**) can be stored in the variable. In fact, the null reference is the default value for a variable that contains an object. In an object diagram, a variable containing the null reference is drawn as a box with a slash; for example:



6.1.3 Classes

Like instance objects, Java classes themselves can have variables (called **class variables**) and methods (called **class methods**). In fact, in many object-oriented languages, classes are themselves a kind of object (usually instances of a class named `Class`!). However, in Java, classes are not objects. They are "second-class citizens" of the Java world. They have names, but there are no references (pointers) to classes. So it is not possible to store a class in a variable, or pass it as a parameter to a method.

In object diagrams, a class named class-name will be depicted as a box labeled class-name class. The box contains the class variables of the class. For example, suppose that the Student class keeps track of the number of students and the last student created. Then a box for the Student class could be added to the above diagram as shown below.



Important: An object diagram should **never** show a reference (pointer) to a class.

(continues…)

Class variables are tagged by the keyword **static** within a class declaration. For example, the above diagram implies that the declaration of the Student class matches the following skeleton:

```
public class Student {
   // Class variables
   private static int num = 0;
   private static Student last = null;

   // Instance variables
   private String firstname;
   private String lastname;
   private int [ ] scores;

   // Constructor
   public Student (String fst, String lst, int [ ] scores) {
      firstname = fst;
      lastname = lst;
      this.scores = scores;
      num = num + 1;
      last = this;
   }

   // Class methods ...

   // Instances methods ...
}
```

7 More Notes on Class Design

7.1 Class Specifications

In the OOP Introduction note, we made the point that one does not need to know every detail about how a car works in order to be a thoroughly competent **user** of a car. In particular, one needs only be familiar with the **public interface** of a car. The **implementation** details are not important to the external user.

The same thing is true in object-oriented programming. For any class in a Java program, we can make a distinction between the public interface of the class and its implementation. The public interface of a class is a subset of all its features that the programmer has chosen to make **public** -- directly accessible from outside the class definition. The public interface is like a **specification** for things the class is able to do. It is also effectively a **contract** between the class in question and other classes that make use of it.

The information hiding that creates this distinction between the public interface and the private implementation is called **encapsulation** -- a key principle of OOP. One of the goals of class design is to decide which features of a class should be part of the public interface and which should remain private -- in other words, to define effective encapsulation for the class.

7.2 Client and Supplier Classes

When program code (in a class, since every Java program is written as a class definition) makes use of another class, that program code is called **client code**, or a **client class**. The class being used is called **supplier code** or a **supplier class**. Examine the following bit of code:

```
public class Sample {
   private String s;
   public Sample() {
      s = "Just Testing";
   }
}
```

What is the name of the client class in this short program? What is the name of the supplier class? The client class is **Sample** and the supplier is **String**. **Sample** is using **String**.

7.3 public vs. private

A client class can make use of any feature of a supplier that is part of the supplier's public interface. The programmer of the supplier class can choose which features will be public or private by using **access modifiers**. The words ***public*** and ***private*** are keywords in the Java language, and are examples of access modifiers.

- Any class element (e.g. instance variable, constructor, method) that is declared ***public*** is available for client code to use (access) directly.
- Class elements declared ***private*** may be used only inside the class definition itself -- they are not directly accessible by client code.

How does a programmer decide what features to make public or private? Here are some good general rules:

- Instance variables should almost always be private. Use public query and/or update methods to provide whatever access is allowed.
- Methods are most often public, but in general only those features that need to be accessed directly by client code should be public. If a method is used only to perform an intermediate calculation or operation, for example, it should probably be private.

We will consider this question regularly as we continue our study of OOP.

8 Ways to Represent a Class

Depending on the information needed, a Java class may be viewed in different ways, from extremely complete to very abstract. Here are the three common ways to view a Java class.

8.1 Source Code (implementation)

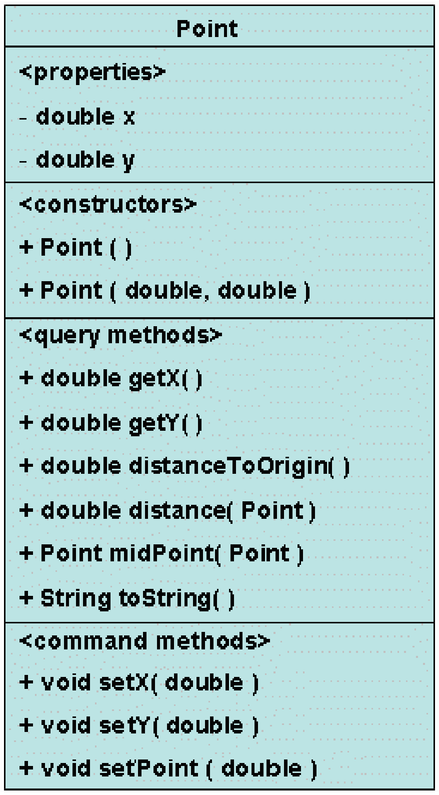
The most detailed look at a class is to examine the source code directly. The source code contains all implementation details and describes exactly the behavior of the class. Of course, there are some drawbacks to using this view of a class:

- source code may not be available to you
- source code is only readable by programmers, and the more complex it is, the more difficult it may be to read

- source code often (usually) contains more detail than you need, and the excess detail may make it harder to extract the essential information
- one final, subtle drawback: making decisions based on the **private**, internal implementation details may backfire, because these details could be changed at some future time; a class only promises to adhere to its public interface (see next section)

Think of the source code for the Point class as an example of this level of detail.

8.2 UML Class Diagram (most abstract)



The figure at right is a class diagram block for version of the Point class provided in this week's sample code. (Note that this version is different from the Point class in the textbook.)

Here are some details to notice:

- The region at the top of the block is reserved for the name of the class only.
- Items below this are typically ordered like this: **properties**, followed by **constructors**, then **methods**. There is no real rule for the ordering of methods.
- A plus sign (**+**) means the member or constructor is ***public***; a minus sign (**-**) means it is ***private***.
- Each method and constructor is generally represented by a **method signature** only: the method's **return type**, **name**, and a list of the **data types of the parameters** in the order that they are declared. Parameter names could also be listed if that information is important.
- It is common for them to leave out details that are not needed in a particular application in order to focus on what is important.

8.3 Javadoc Specification (public interface - abstract)

Every class should come with a **specification** that defines its **public interface**.

The Javadoc specification for the Point class is a good example of a public interface specification. In particular, notice that the instance variables are not shown in this specification because they are private.

8.3.1 Javadoc Comments

It is important to have a fully-documented public interface for every class. You can see an example of this by examining the documentation for class Point. This very complete and nicely-formatted documentation was generated automatically from the code and Javadoc comments written in the Point class. Take some time to examine the source code for class Point and the documentation side-by-side. See if you can see how information from the Javadoc comments made its way into the documentation.

8.3.2 Java API Documentation

The Java API (Application Programming Interface) is a large library of pre-defined classes that comes as part of the Java SDK (Software Development Kit) that forms the foundation of our development environment. These classes are organized into a series of packages. Complete class specifications (public interface descriptions) are provided for every class in the **Java API Documentation**. By now you have probably figured out that this documentation is automatically generated from Javadoc comments included in the actual code that defines these classes.

Take a few minutes now to take another look at the documentation for class **String**. See if you can improve your understanding of what the **String** class is capable of doing for you. What methods do you see listed? Can you think of situations where some of them might be useful. One valuable thing you can do as you learn the Java language is try to develop a solid understanding of **what** kinds of things can be done with the tools in the library. Don't worry about **how** they are done -- you can always look up the details if you ever need them. That's what the documentation is for!

9 Keyword static

We've seen the keyword static used in 2 different ways:

1. a static method -- this method gets all the data it needs through the parameter list and typically returns a value (or displays to the terminal).
2. static final constants

It's time to talk a bit more about this keyword and understand its capabilities

We've learned about instance variables -- each instance of a class has its own version of these attributes. Instance methods are used to access/modify these instance variables.

A **static variable**, also known as a **class variable**, is one that belongs to the class as a whole. There is only 1 copy of a static variable, and that is shared by all objects of that class. Think about the BankAccount class for example. What if we wanted to modify it so that our program could keep track of how many BankAccount objects had been created? We would need a counter variable, but we only need one; one that is shared by all instances. So we can create a static variable:

```
public class BankAccount {
   private static int counter = 0;
   // the instance variables, etc
   ...
}
```

We still have the choice to make it public or private. Notice that we initialize it as it is declared. We don't initialize static variables in constructors (otherwise, this variable would keep getting re-initialized every time a new instance is created.)

What we do want is for this variable to be updated whenever a new BankAccount is created. We can do that by incrementing it in the constructor:

```
public class BankAccount {
   private static int counter = 0;
   // the instance variables, etc
   ...

   public BankAccount(String name, String ID, double newBalance){
      // initialize instance variables as before
      ...
      counter++;
   }
}
```

Remember, there is only 1 counter variable, shared by all BankAccount objects (as opposed to the balance instance variable, where each object has its own).

Finally, clients will want to access this class variable just as they access instance variables. If instance methods access instance variables, what type of method accesses static variables? You got it, static methods:

```
public class BankAccount {
   private static int counter = 0;
   // the instance variables, etc
   ...

   public BankAccount(String name, String ID, double newBalance){
      // initialize instance variables as before
      ...
      counter++;
   }

   public static int getCount(){
      return counter;
   }
}
```

So here's another reason we design static methods -- to allow clients to access/mutate static (class) variables.

Let's revisit our static final constants. There are 2 keywords here and they each do a different job:

- static means that we only have one copy of this variable, to be shared by all objects of this class
- final means that once the variable is declared and initialized, its value can never change.