
Chapter 3

Defining Object Behavior

What is in This Chapter ?

This chapter discusses the basic idea behind object-oriented programming... that of **defining objects** in terms of their **state** and **behavior**. It revisits the notion of **constructors** and then explains how functions and procedures (also called **methods**) can be implemented and associated with an object's definition. The difference between **instance methods** and **static/class methods** is discussed. The notion of **encapsulation** is introduced as well as the **toString()** method that affects how an object appears when printed. The chapter concludes with a **Bank example** that makes use of 4 different objects.



3.1 Object Constructors (Re-Visited)

A constructor is a special chunk of code that we can write in our object classes that will allow us to **hide the ugliness** of setting all of the initial values for our objects each time we use them. The main advantage of making a constructor is that it will **allow us to reduce the amount of code that we need to write each time we make a new object**.

Consider, for example, a **Person** which is defined as shown below with 6 attributes:

```
public class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;
}
```

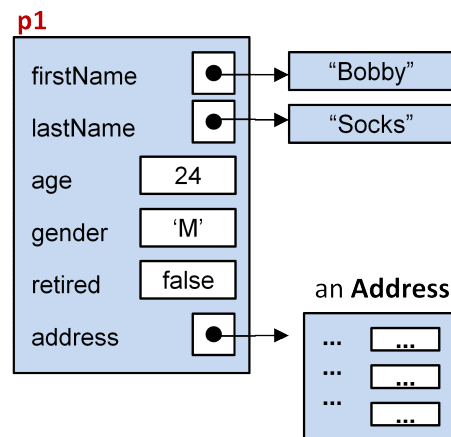
We can create a new **Person** object as follows: `new Person()`

However, to set the values for the person, we would need multiple lines of code:

```
Person    p1;

p1 = new Person();

p1.firstName = "Bobby";
p1.lastName = "Socks";
p1.age = 24;
p1.gender = 'M';
p1.retired = false;
p1.address = new Address();
```



We can write a **constructor** for this class that allows us to provide initial values for all of the object's attributes. A constructor is a special kind of function:

```
public Person(String f, String l, int a, char g, boolean r, Address d) {
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = r;
    address = d;
}
```

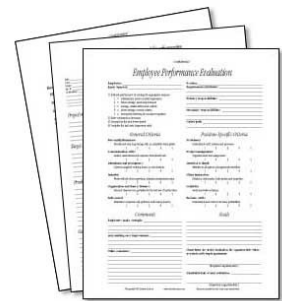
Notice in JAVA, we usually indicate that a constructor is public by placing the **public** keyword in front of it. By defining the above constructor, we are thus able to specify the initial parameters for any newly-created **Person** objects as follows:

```
Person    p1, p2, p3;

p1 = new Person("Bobby", "Socks", 24, 'M', false, anAddress);
p2 = new Person("Holly", "Day", 72, 'F', true, anotherAddress);
p3 = new Person("Hank", "Urchif", 19, 'M', false, yetAnotherAddress);
```

Certainly, constructors allow us to greatly simplify our code when we need to create objects in our program.

Suppose though, that we do not know the initial parameter values to use. This would be analogous to the situation in real life where someone fills out a form but leaves some information blank. What do we do when the person leaves out information? We have two possible choices. Either **(1)** do not let them leave out any information, or **(2)** choose some kind of “default” values for the blank parts (i.e., make some assumptions by filling in something appropriate).



The above constructor forces the user of our objects to supply parameters for ALL of the instance variables when they use (i.e., call) our constructor ... which is approach number (1) above. However, in JAVA, we are allowed to create more than one constructor as long as the constructors each have a unique list of parameter types.

What if, for example, we did not know the person's age, nor their address.

```
Person    p1, p2;

p1 = new Person("Hank", "Urchif", ?, 'M', false, ?);
p2 = new Person("Holly", "Day", ?, 'F', true, ?);
```

For this situation, we can actually define a second constructor that leaves out these two parameters:

```
public Person(String f, String l, char g, boolean r) {
    firstName = f;
    lastName = l;
    gender = g;
    retired = r;
    age = 0;
    address = null;
}
```

Notice that there are two less parameters now (i.e., no age and no address). However, you will notice that we still set the age and address to some *default* values of our choosing. What is a good default age and address ? Well, we used **0** and **null**. Since we do not have an **Address** object to store in the address instance variable, we leave it undefined by setting it to **null**. Alternatively, we could have created a “dummy” **Address** object with some kind of values that would be recognizable as invalid such as:

```
address = new Address();
```

It is entirely up to you to decide what the default values should be. Make sure not to pick something that may be mistaken for valid data. For example, some bad default values for firstName and lastName would be “John” and “Doe” because there may indeed be a real person called “John Doe”.

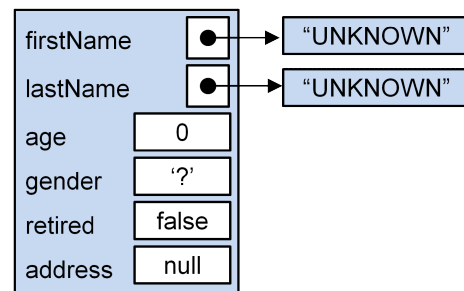
Here is one more constructor that takes no parameters. It has a special name and is known as the **zero-parameter constructor**, the **zero-argument constructor** or the **default constructor**. This time there are no parameters at all, so we need to pick default values for all the attributes:

```
public Person() {
    firstName = "UNKNOWN";
    lastName = "UNKNOWN";
    gender = '?';
    retired = false;
    age = 0;
    address = null;
}
```

You can actually create as many constructors as you want. You just need to write them all one after each other in your class definition and the user can decide which one to use at any time. Here is our resulting **Person** class definition showing the four constructors ...

```
public class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;

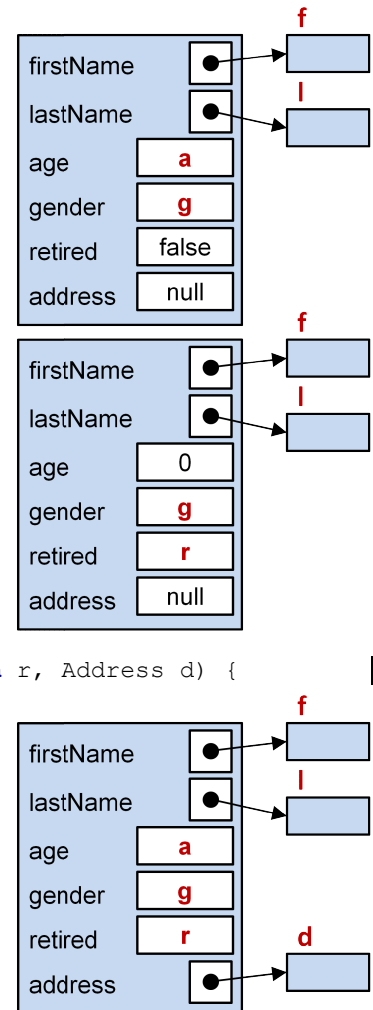
    // This is the zero-parameter constructor
    public Person() {
        firstName = "UNKNOWN";
        lastName = "UNKNOWN";
        gender = '?';
        retired = false;
        age = 0;
        address = null;
    }
}
```



```
// This is a 4-parameter constructor
public Person(String f, String l, int a, char g) {
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = false;
    address = null;
}

// This is another 4-parameter constructor
public Person(String f, String l, char g, boolean r) {
    firstName = f;
    lastName = l;
    gender = g;
    retired = r;
    age = 0;
    address = null;
}

// This is a 6-parameter constructor
public Person(String f, String l, int a, char g, boolean r, Address d) {
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = r;
    address = d;
}
}
```



At any time we can use any of these constructors:

```
Person    p1, p2, p3, p4;

p1 = new Person();
p2 = new Person("Sue", "Purmann", 58, 'F');
p3 = new Person("Holly", "Day", 'F', true);
p4 = new Person("Hank", "Urchif", 19, 'M', false, new Address(...));
```

Note that it is always a good idea to ensure that you have a zero-parameter constructor. As it turns out, if you do not write any constructors, JAVA provides a zero-parameter constructor for free. That is, we can always say `new Car()`, `new Person()`, `new Address()`, `new BankAccount()` etc.. without even writing those constructors. However, once you write a constructor that has parameters, the free zero-parameter constructor is no longer available. That is, for example, if you write constructors in your **Person** class that all take one or more parameters, then you will no longer be able to use `new Person()`. JAVA will generate an error saying:

```
cannot find symbol constructor Person()
```

In general, you should always make your own zero-parameter constructor along with any others that you might like to use because others who use your class may expect there to be a zero-parameter constructor available.

3.2 Defining Methods

At this point you should understand that objects are used to group variables together in order to represent something called a *data structure*. Each object therefore has a set of *attributes* (also called *instance variables*) that represent the differences between members of the same class. For example, a **Vehicle** object may define a **color** attribute ... that is ... each vehicle has a color. However, that **color** value can vary from vehicle to vehicle:



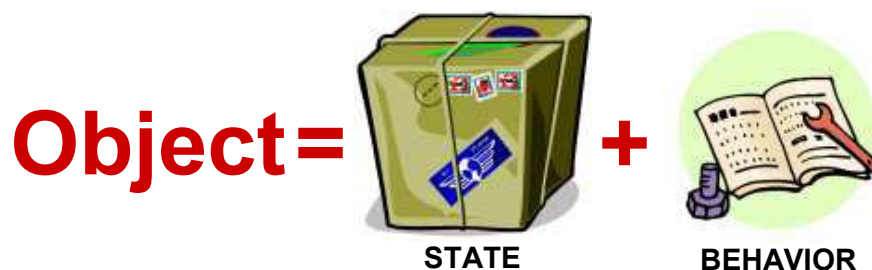
However, in real life, vehicles also vary in terms of their performance characteristics, their functionality, their abilities and their features/options:



Likewise, in object-oriented programming, in addition to defining attributes, we can also define how one particular kind of object's performance and behaviors differ from another's. Defining an object's behavior is as simple as deciding what kind of functionality that the object should have. This is nothing more than deciding which functions or procedures are required to access, modify or compute information based on the object's attributes.

Simply put ... when we define an object, we

- (1) define its attributes
- (2) define the functions and procedures that work on/with the object



What does all of this mean ? Instead of writing a single program with a list of functions/procedures, we will now be associating some of the functions/procedures with various objects. So, we will actually move some of the functions/procedures into our various class definitions.

For example, consider code that causes two cars to accelerate across the screen. We could create a **Car** class which maintains a car's location and speed.

```
public class Car {
    int    x, y;
    float  speed;
}
```

We could then write a test program with procedures that cause a car to be drawn on the screen or to move across the screen. Then at any time, we could cause the car to be repeatedly moved and drawn by calling these procedures.

Notice that the **draw()** and **move()** procedures take (as an incoming parameter) the **Car** object that is to be drawn or moved. This is the object that gets affected by the procedure call. So, in a way, the procedure represents a behavior for the car, as it will affect/modify the **Car** object that is passed in.

```
public class CarTestProgram {

    public static void draw(Car aCar) {
        /* code not shown */
    }
    public static void move(Car aCar) {
        /* code not shown */
    }

    public static void main(String[] args) {
        Car    myCar = new Car();
        Car    yourCar = new Car();

        draw(myCar);
        move(myCar);
        draw(yourCar);
        move(yourCar);
    }
}
```

However, when doing object-oriented programming, we give an object behavior by defining procedures within its class. So, we would define the **Car** class as shown here. Notice that the **draw()** and **move()** procedures are now written in the **Car** class ... that is ... each **Car** object now knows how to move and draw itself. Notice as well that there are no parameters now in the two procedures. That is because these procedures are written within the Car class definition itself, so they will be applied to the Car that they are called upon.

```
public class Car {
    int    x, y;
    float  speed;

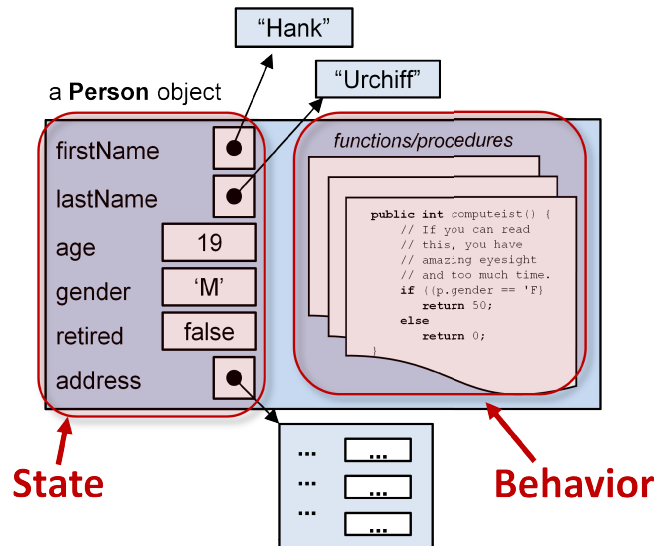
    public void draw() {
        /* code not shown */
    }
    public void move() {
        /* code not shown */
    }
}
```

In the test program, notice that the procedures are no longer here. Since they are in the **Car** class now, we call/access them by using the dot operator (which means "go inside"). So, we tell JAVA to go inside the **Car** class to find the **draw()** and **move()** procedures.

```
public class CarTestProgram {
    public static void main(String[] args) {
        Car    myCar = new Car();
        Car    yourCar = new Car();

        myCar.draw();
        myCar.move();
        yourCar.draw();
        yourCar.move();
    }
}
```


We will write most of our functions and procedures (now referred to as **methods**) within our class definitions along with the attributes. So we can think of an object as being a set of attributes (i.e., representing the object's state) as well as a set of methods (i.e., representing the object's behavior) all included "inside" the class:



Example:

Consider the **Person** class. Assume that we want to write a function that computes and returns the discount for a person who attends the theatre on "Grandma/Granddaughter Night". Assume that the discount should be 50% for women who are retired or to girls who are 12 and under. For all other people, the discount should otherwise be 0%. If we had the **Person** passed in as a parameter to the function, we could write this code:

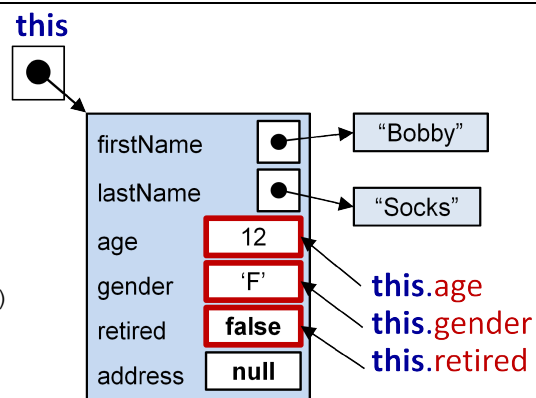
```
public int computeDiscount(Person p) {
    if ((p.gender == 'F') && (p.age < 13 || p.retired))
        return 50;
    else
        return 0;
}
```



To write this as a method in JAVA, we would place this method in the **Person** class after the instance variables and constructors are defined:

```
public class Person {
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public int computeDiscount() {
        if ((this.gender == 'F') &&
            (this.age < 13 || this.retired))
            return 50;
        return 0;
    }
}
```



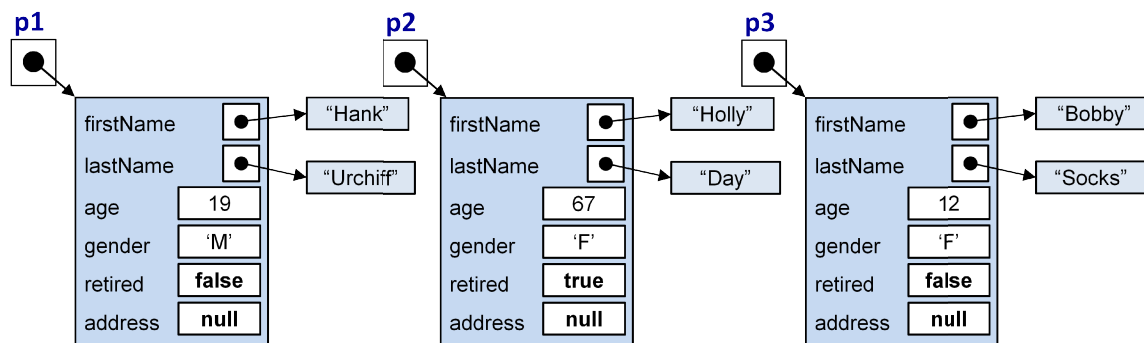
Notice that the **Person** parameter is no longer there and that the word **this** is now being used in place of that parameter. The word **this** is a special word in JAVA that can be used in methods (and constructors) to represent the object that we are applying the behavior to. That is, whatever object that we happen to call the method on, that object is represented by the word **this** within the method's body. You can think of the word **this** as being a *nickname* for the object being "worked on" within the method. Outside of the method, the word **this** is actually undefined (and therefore unusable outside of the method) .

So, if we called the **computeDiscount()** method for different **Person** objects, **this** would represent the different objects **p1**, **p2** and **p3**, each time the method is called, respectively:

```
Person    p1, p2, p3;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);
p3 = new Person("Bobby", "Socks", 12, 'F');

System.out.println("p1's discount = " + p1.computeDiscount());
System.out.println("p2's discount = " + p2.computeDiscount());
System.out.println("p3's discount = " + p3.computeDiscount());
```



As it turns out, if you leave off the keyword **this**, JAVA will "assume" that you meant the object that received the method call in the first place and will act accordingly. Therefore, the following code is equivalent and often preferred since it is shorter:

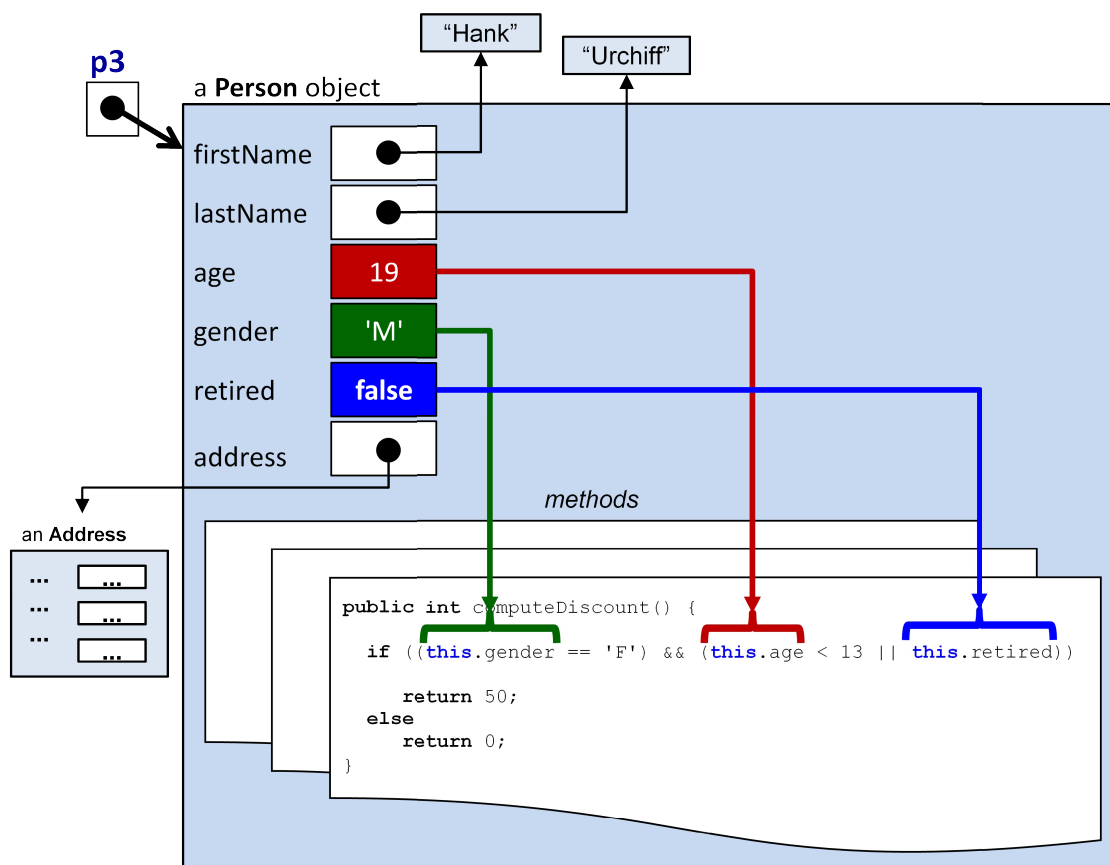
```
public class Person {
    ....

    public int computeDiscount() {
        if ((gender == 'F') && (age < 13 || retired))
            return 50;
        else
            return 0;
    }
}
```

It is important for you to understand that the gender, age and retired attributes are obtained from the **Person** object on which we called the **computeDiscount()** method.

You may have also noticed that the method was declared as **public**. This allows any code outside of the class to use the method.

When we test the method using `p3.computeDiscount()` ... this is a picture of what is happening inside the object:



Consider writing another method that determines whether one person is older than another person. We can call the method **isOlderThan(Person x)** and have it return a **boolean** value:

```
public boolean isOlderThan(Person x) {
    if (this.age > x.age)
        return true;
    else
        return false;
}
```



... or the more efficient version:

```

public boolean isOlderThan(Person x) {
    return (this.age > x.age);
}

```

Here is a portion of a program that determines the oldest of 3 people:

```

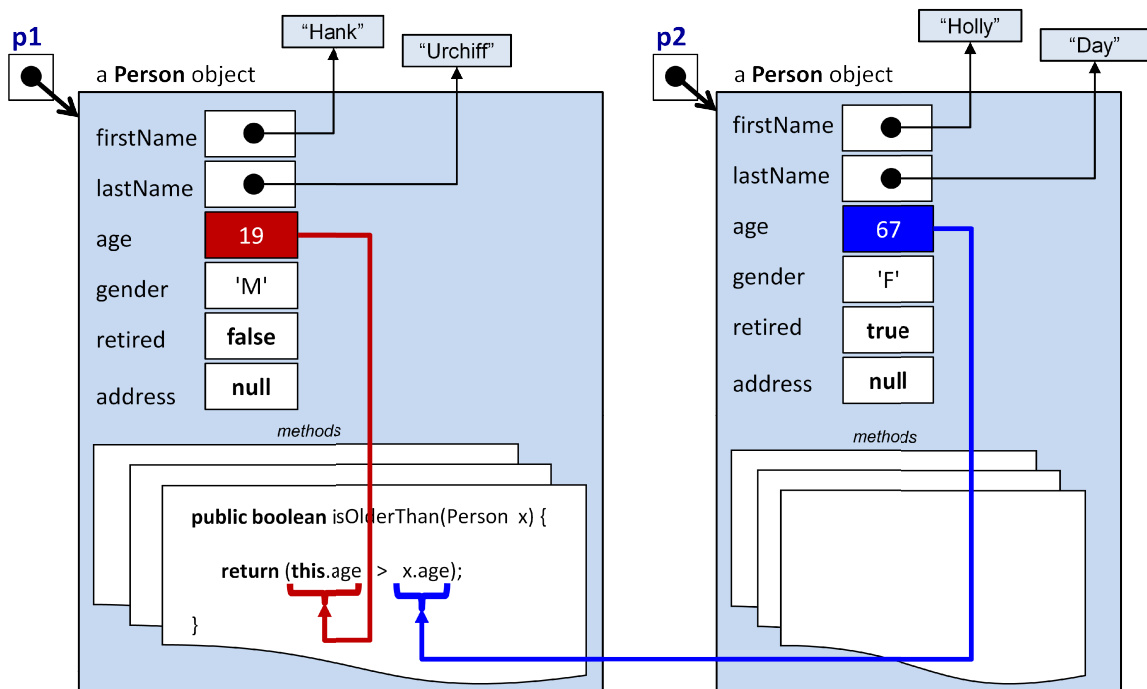
Person    p1, p2, p3, oldest;

p1 = new Person("Hank", "Urchiff", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);
p3 = new Person("Bobby", "Socks", 12, 'F');

if (p1.isOlderThan(p2) && p1.isOlderThan(p3))
    oldest = p1;
else if (p2.isOlderThan(p1) && p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;

```

Consider what happens inside **p1** as we call **p1.isOlderThan(p2)**:



The method accesses the age that is stored within both **Person** objects **this** and **x**.

How could we write a similar method called **oldest()** that returns the oldest of the two **Person** objects, instead of just returning a boolean ?

```
public Person oldest(Person x) {
    if (this.age > x.age)
        return this;
    else
        return x;
}
```



Notice how the code is similar except that it now returns the **Person** object instead. Now we can simplify our program that determines the oldest of 3 people:

```
Person    p1, p2, p3, oldest;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);
p3 = new Person("Bobby", "Socks", 12, 'F');

oldest = p1.oldest(p2.oldest(p3));
```

Do you understand how this code works ? Notice how the innermost **oldest()** method returns the oldest of the **p2** and **p3**. Then this oldest one is compared with **p1** in the outermost **oldest()** method call to find the oldest of the three.

In addition to writing such functions, we could write procedures that simply modify the object. For example, if we wanted to implement a **retire()** method that causes a person to retire, it would be straight forward as follows:

```
public void retire() {
    this.retired = true;
}
```

Notice that the code simply sets the retired status of the person and that the method has a void return type, indicating that there is no "answer" returned from the method's computations.

How about a method to swap the names of two people ?

```
public void swapNameWith(Person x) {
    String tempName;

    // Swap the first names
    tempName = this.firstName;
    this.firstName = x.firstName;
    x.firstName = tempName;

    // Swap the last names
    tempName = this.lastName;
    this.lastName = x.lastName;
    x.lastName = tempName;
}
```



Notice how the temporary variable is required to store the **String** that is being replaced.

At this point let's step back and see what we have done. We have created 5 interesting methods (i.e., behaviors) for our **Person** object (i.e., **computeDiscount()**, **isOlderThan()**, **oldest()**, **retire()** and **swapNameWith()**). All of these methods were written one after another within the class, usually after the constructors. Here, to the right, is the structure of the class now as it contains all the methods that we wrote (the method code has been left blank to save space).

Now although these methods were **defined** in the class, they are not **used** within the class. We wrote various pieces of test code that call the methods in order to test them. Here is a more complete test program that tests all of our methods in one shot:

```
public class Person {
    // These are the instance variables
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean    retired;
    Address   address;

    // These are the constructors
    public Person() { ... }
    public Person(String fn, ...) { ... }

    // These are our methods
    public int computeDiscount() { ... }
    public boolean isOlderThan(Person x) { ... }
    public Person oldest(Person x) { ... }
    public void retire() { ... }
    public void swapNameWith(Person x) { ... }
}
```

```
public class FullPersonTestProgram {
    public static void main(String args[]) {
        Person  p1, p2, p3;

        p1 = new Person("Hank", "Urchif", 19, 'M');
        p2 = new Person("Holly", "Day", 67, 'F', true, null);
        p3 = new Person("Bobby", "Socks", 12, 'F');

        System.out.println("The discount for Hank is " +
                           p1.computeDiscount());
        System.out.println("The discount for Holly is " +
                           p2.computeDiscount());
        System.out.println("The discount for Bobby is " +
                           p3.computeDiscount());

        System.out.println("Is Hank older than Holly ? ..." +
                           p1.isOlderThan(p2));
        System.out.println("The oldest person is " +
                           p1.oldest(p2.oldest(p3)).firstName);

        System.out.println("Holly is retired ? ... " + p2.retired);
        p2.retire();
        System.out.println("Holly is retired ? ... " + p2.retired);
        p2.swapNameWith(p3);
        System.out.println("Holly's name is now: " +
                           p2.firstName + " " + p2.lastName);
        System.out.println("Bobby's name is now: " +
                           p3.firstName + " " + p3.lastName);
    }
}
```

Here is the output:

```
The discount for Hank is 0
The discount for Holly is 50
The discount for Bobby is 50
Is Hank older than Holly ? ...false
The oldest person is Holly
Holly is retired ? ... false
Holly is retired ? ... true
Holly's name is now: Bobby Socks
Bobby's name is now: Holly Day
```

3.3 Null Pointer Exceptions

In regards to calling methods, we must make sure that the object whose method we are trying to call has been through the construction process. For example, consider the following code:

```
Person    p;

System.out.println(p.computeDiscount());
```



This code will not compile. JAVA will give a compile error for the second line of code saying:

variable p might not have been initialized

JAVA is trying to tell you that you forgot to give a value to the variable **p**. In this case, we forgot to create a **Person** object.

Assume then that we created the **Person** as follows and then tried to get the streetName:

```
Person    p;

p = new Person("Hank", "Urchif", 'M', false);
System.out.println(p.address.streetName);
```

This code will now compile. Assume that the **Person** class was defined as follows:

```
public class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;
```

```
public Person(String fn, String ln, char g, boolean r) {  
    firstName = fn;  
    lastName = ln;  
    gender = g;  
    retired = r;  
    age = 0;  
    address = null;  
}  
...  
}
```

Here the **address** attribute stores an **Address** object which is assumed to have an instance variable called streetName.

What will happen when we do this:

```
p.address.streetName
```

The code will generate a **java.lang.NullPointerException**. That means, JAVA is telling you that you are trying to do something with an object that was not yet defined. Whenever you get this kind of error, look at the line of code on which the error was generated. The error is always due to something in front of a dot **.** character being **null** instead of being an actual object. In our case, there are two dots in the code on that line. Therefore, either **p** is **null** or **p.address** is **null**, that is the only two possibilities. Well, we are sure that we assigned a value to **p** on the line above, so then **p.address** must be **null**. Indeed that is what has happened, as you can tell from the constructor.

To fix this, we need to do one of three things:

1. Remove the line that attempts to access the streetName from the address, and access it later in the program after we are sure there is an address there.
2. Check for a **null** before we try to print it and then don't print if it is **null** ... but this may not be desirable.
3. Think about *why* the address is **null**. Perhaps we just forgot to set it to a proper value. We can make sure that it is not **null** by giving it a proper value before we attempt to use it.

NullPointerExceptions are one of the most common errors that you will get when programming in JAVA. Most of the time, you get the error simply because you forgot to initialize a variable somewhere (i.e., you forgot to create a new object and store it in the variable).

3.4 Overloading

When we write two methods in the same class with the same name, this is called **overloading**. Overloading is only allowed if the similar-named methods have a **different** set of parameters. Normally, when we write programs we do not *think* about writing methods with the same name ... we just do it naturally. For example, imagine implementing a variety of **eat()** methods for the **Person** class as follows:

```
public void eat(Apple x) { ... }
public void eat(Orange x) { ... }
public void eat(Banana x, Banana y) { ... }
```

Notice that all the methods are called **eat()**, but that there is a variety of parameters, allowing the person to eat either an Apple, an Orange or two Banana objects. Imagine the code below somewhere in your program that calls the **eat()** method, passing in an object **z** of some type:

```
Person p;

p = new Person();
p.eat(z);
```



How does JAVA know which of the 3 **eat()** methods to call? Well, JAVA will look at what kind of object **z** actually is. If it is an **Apple** object, then it will call the 1st **eat()** method. If it is an **Orange** object, it will call the 2nd method. What if **z** is a Banana? It will NOT call the 3rd method ... because the 3rd method requires 2 Bananas and we are only passing in one. A call of **p.eat(z, z)** would call the 3rd method if **z** was a Banana. In all other cases, the JAVA compiler will give you an error stating:

```
cannot find symbol method eat(...)
```

where the **...** above is a list of the types of parameters that you are trying to use.

JAVA will NOT allow you to have two methods with the **same name AND parameter types** because it cannot distinguish between the methods when you go to use them. So, the following code will not compile:

```
public double calculatePayment(BankAccount account){...}
public double calculatePayment(BankAccount x){...}
```

You will get an error saying:

```
calculatePayment(BankAccount) is already defined in Person
```

Recall our method called **isOlderThan()** that returns a **boolean** indicating whether or not a person is older than the one passed in as a parameter:

```
public boolean isOlderThan(Person x) {
    return (this.age > x.age);
}
```

We could actually write another method in the **Person** class that took two **Person** objects as parameters:

```
public boolean isOlderThan(Person x, Person y) {
    return (this.age > x.age) && (this.age > y.age);
}
```

... and even a third method with 3 parameters:

```
public boolean isOlderThan(Person x, Person y, Person z) {
    return (this.age > x.age) && (this.age > y.age) && (this.age > z.age);
}
```

As a result, we could use any of these methods in our program:

```
Person    p1, p2, p3, p4, oldest;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F');
p3 = new Person("Bobby", "Socks", 12, 'M');
p4 = new Person("Sue", "Purmann", 58, 'F');

if (p1.isOlderThan(p2,p3,p4))
    oldest = p1;
else if (p2.isOlderThan(p3,p4))
    oldest = p2;
else if (p3.isOlderThan(p4))
    oldest = p3;
else
    oldest = p4;
```

Keep in mind, however, that the parameters need not be the same type. You can have any types of parameters. Remember as well that the order makes a difference. So these would represent unique methods:

```
public int computeHealthRisk(int age, int weight, boolean smoker) { ... }
public int computeHealthRisk(boolean smoker, int age, int weight) { ... }
public int computeHealthRisk(int weight, boolean smoker, int age) { ... }
```

But these two cannot be defined together in the same class because the parameter **types** are in the same order:

```
public int computeHealthRisk(int age, int weight, boolean smoker) { ... }
public int computeHealthRisk(int weight, int age, boolean smoker) { ... }
```

3.5 Instance vs. Class (i.e., static) Methods

The methods that we have written so far have defined behaviors that worked on specific object instances. For example, when we used the **computeDiscount()** method, we did this:

```
Person    p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F');

System.out.println("p1's discount = " + p1.computeDiscount());
System.out.println("p2's discount = " + p2.computeDiscount());
```

In this example, **p1** and **p2** are variables that store instances of the **Person** class (i.e., specific individual **Person** objects). Therefore, the **computeDiscount()** method is considered to be an **instance method** of the **Person** class, since it operates on a specific *instance* of the **Person** class.

Instance methods represent behaviors (functions and procedures) that are to be performed on the particular object that we called the method for (i.e., the receiver of the method).

Instance methods typically access the inner parts of the receiver object (i.e., its attributes) and perform some calculation or change the object's attributes in some way.

A method that does not require an instance of an object to work is called a **class method**:

Class methods represent behaviors (functions and procedures) that are performed on a class ... without a particular object in mind.

Therefore, class methods do not represent a behavior to be performed on a particular receiver object. Instead, a class method represents a general function/procedure that simply happens to be located within a particular class, but does not *necessarily* have anything to do with instances of that class. Generally, class methods are not used to modify a particular instance of a class, but usually perform some computation.

For example, recall the code for the **computeDiscount()** method:

```
public int computeDiscount() {
    if ((this.gender == 'F') && (this.age < 13 || this.retired))
        return 50;
    else
        return 0;
}
```

We could have re-written the **computeDiscount()** method by supplying the appropriate **Person** object as a parameter to the method as follows:

```
public int computeDiscount(Person p) {
    if ((p.gender == 'F') && (p.age < 13 || p.retired))
        return 50;
    else
        return 0;
}
```

Notice how the method now accesses the person **p** that is passed in as the parameter, instead of the receiver **this**. If we do this, the code result is now fully-dependent on the attributes of the incoming parameter **p**, and hence independent of the receiver altogether. To call this method, we would need to pass in the person as a parameter:

```
Person    p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);

System.out.println("p1's discount = " + p1.computeDiscount(p1));
System.out.println("p2's discount = " + p2.computeDiscount(p2));
```

We would never do this, however, since within the **computeDiscount()** method, the parameter **p** and the keyword **this** both point to the same **Person** object. So, the extra parameter is not useful since we can simply use **this** to access the person. Instead, what we probably wanted to do is to make a general function that can be written anywhere (i.e., in any class) that allows a discount to be computed for any **Person** object that was passed in as a parameter. Consider this class for example:

```
public class Toolkit {
    ...
    public int computeDiscount(Person p) {
        if ((p.gender == 'F') && (p.age < 13 || p.retired))
            return 50;
        else
            return 0;
    }
    ...
}
```

Now to call the method, we would need to make an instance of **Toolkit** as follows:

```
new Toolkit().computeDiscount(p1);
```

But this seems awkward. If we wanted to use this "tool-like" function on many people, we could do this:

```

Person    p1, p2, p3;
Toolkit   toolkit;

toolkit = new Toolkit();
p1 = ...;
p2 = ...;
p3 = ...;
System.out.println(toolkit.computeDiscount(p1));
System.out.println(toolkit.computeDiscount(p2));
System.out.println(toolkit.computeDiscount(p3));

```

Now we can see that **toolkit** is indeed a separate class from **Person** and that it acts as a container that holds on to the useful **computeDiscount()** function. However, we can simplify the code.

Anytime that we write a method that does not modify or access the attributes of an instance of the class that it is written in, the method functionality does not change. In other words, the code is not changing from instance to instance ... and is therefore considered **static**. In our example, the code inside of the **computeDiscount()** method does not access or modify and attributes of the **Toolkit** class ... it simply accesses the attributes of the **Person** passed in as a parameter as performs a computation. Therefore this method should be made **static**.

How do we do this? We simply add the **static** keyword in front of the method definition:

```

public class Toolkit {
    ...
    public static int computeDiscount(Person p) {
        if ((p.gender == 'F') && (p.age < 13 || p.retired))
            return 50;
        else
            return 0;
    }
    ...
}

```

Now we do not need to make a *new Toolkit* object in order to call the method. Instead, we simply use the **Toolkit** class name to call the method. Here is how the code changes. Notice how much simpler it is to use the method once it has been made **static**. (to save space, `System.out.println` has been written as **S.o.p** below):

Using it as an <i>instance</i> method	Using it as a <i>class</i> method
<pre> Person p1, p2, p3; Toolkit toolkit; toolkit = new Toolkit(); p1 = ...; p2 = ...; p3 = ...; S.o.p(toolkit.computeDiscount(p1)); S.o.p(toolkit.computeDiscount(p2)); S.o.p(toolkit.computeDiscount(p3)); </pre>	<pre> Person p1, p2, p3; p1 = ...; p2 = ...; p3 = ...; S.o.p(Toolkit.computeDiscount(p1)); S.o.p(Toolkit.computeDiscount(p2)); S.o.p(Toolkit.computeDiscount(p3)); </pre>

This is the essence of a class/static method ... the idea that the method does not necessarily need to be called by using an instance of the class.

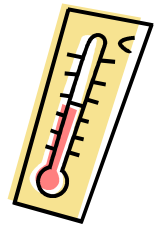
Hopefully, you will have noticed that the main difference between an instance method and a class/static method is simply in the way in which it is called. To repeat ... *instance methods* are called by supplying a specific instance of the object in front of the method call (i.e., a variable of the same type as the class in which the method is defined in), while *class methods* supply a class name in front of the method call:

```
// calling an instance method...
variableOfTypeX.instanceMethodWrittenInClassX(...);

// calling a class method...
ClassNameY.staticMethodWrittenInClassY(...);
```

Often, we use class methods to write functions that may have nothing to do with objects at all. For example, consider methods that convert a temperature value from centigrade to fahrenheit and vice-versa:

```
public static double centigradeToFahrenheit(double temp) {
    return temp * (9.0 / 5.0) + 32.0;
}
public static double fahrenheitToCentigrade(double temp) {
    return 5.0 * (temp - 32.0) / 9.0;
}
```



Where do we write such methods since they only deal with primitives, not objects? The answer is ... we can write them anywhere. We can place them at the top of the class that we would like to use them in. Or ... if these functions are to be used from multiple classes in our application, we could make another tool-like class and put them in there:

```
public class ConversionTools {
    ...
}
```

Then we could use it as follows:

```
double f = ConversionTools.centigradeToFahrenheit(18.762);
```

As you browse through the JAVA class libraries, you will notice that there are some useful static methods, however ... most methods that you will write for your own objects will be instance methods.

3.6 Encapsulation - Protecting An Object's Internals

When creating and defining an object it is a good idea to keep it simple so that anybody who uses that object in the future (including yourself) can remember how to use it. Often, there are details about an object that we don't need to know about in order to use the object. For example, when we drive a car, we need to know simple things such as:

- starting/ stopping
- steering
- changing gears
- braking, etc..



However, we do not need to worry about things such as:

- assembling the carburetor
- adjusting the spark plug timing
- installing gas lines
- changing the muffler, etc..



Cars are clearly designed to be easy to drive, requiring a simple and easy-to-understand interface. Similarly, it is important that we make our code easy to use and easy to understand. Otherwise, making changes to the code, debugging it and extending it with new features can quickly become very difficult and time consuming.

In order to keep our code simple, we need to make the interface (or "outside view") of our objects as simple as possible. That means, we need to "**hide the details**" of our object that most people would not need to worry about. That is, we need to hide some of the attributes (complicated parts) and methods (complicated procedures) for our object "under the hood", so to speak.



In addition to simplicity, there is another reason to hide some of the details of our object. We would like to prevent outsiders from "messing around with" the inner details of an object. For example we lock our car doors and trunk so that people don't get in there and take things away or damage them etc.. Similarly, for example, if we allow anyone to access the attributes of our object and perform behaviors on it in the wrong order, then this could lead to corrupt data and/or various types of errors in our code.

The idea of hiding the unnecessary details of an object and protecting inner parts of that object from general users is called *encapsulation*:

Encapsulation involves enclosing an object with a kind of "protective bubble" so that it cannot be accessed or modified without proper permission.



In JAVA, we protect and hide attributes and behaviors by using something called an **access modifier**.

*An **access modifier** is a permission setting for our attributes and methods so that they will be visible/modifiable/usable from some places in our code but not from other places.*

Access modifiers are like access levels in a high security building (e.g., No access, Level 1 access, Level 2 access, etc..)



By using access modifiers properly, when working with a team of software developers on a large program, some developers will have the freedom to access or modify attributes or methods from various objects, while others will not be allowed such freedom to view or change portions the objects as they would like to.

We have already been using an access modifier called **public** when we wrote our classes, constructors and various methods:

```
public class Person { ... }  
public Person(String firstName, ...) { ... }  
public static void main(String[] args) { ... }  
public int computeDiscount() { ... }  
public void deposit() { ... }
```

The keyword **public** at the front of a method declaration means that the method is publicly available to everyone, so that these methods may be called from anywhere.

For most classes, constructors and methods, we do not need to write **public**. If we leave off this access modifier, then the class/constructor/method will have what is known as **default access** ... meaning that the methods may be called from any code that is in the same package or folder that this method's class is saved into. If we write all of our code in the same folder, then **default** and **public** access means the same thing.

There are two other access modifier options available called **private** and **protected**. When we declare a method as **private**, we would not be able to use this method from any class other than the class in which it is defined. **Protected** methods are methods that may be called from the method's own class or from one of its subclasses (more on this soon). So here is a summary of the access modifiers for methods:

- none - can be called from any class in the same folder
- **public** - can be called from anywhere
- **private** - can only be called from this class
- **protected** - can be called from this class or any subclasses (discussed later)

In this course, most of the methods that we write are **public** methods which allows the most freedom to access and modify our objects. Usually, **private** methods are known as **helper methods** since they are often created for the purpose of helping to reduce the size of a larger **public** method or when a piece of code is shared by several methods.

For example, consider bringing in your car for repair. The publicly available method would be called to **repair()** the car. However, many smaller sub-routines are performed as part of the repair process (e.g., **runDiagnostics()**, **disassembleEngine()** etc...). From the point of view of the user of the class, there is no need to understand the inner workings of the repair process. The user of the class may simply need to know that the car can be repaired, regardless of how it is done. Here is an example of breaking up the repair problem into *helper* methods that do the sub-routines as part of the repair ...

```
public class Car {
    public void repair() {
        this.runDiagnostics();
        this.disassembleEngine();
        this.repairBrokenParts();
        this.reassembleEngine();
        this.runDiagnostics();
    }
    private void runDiagnostics() { /*...*/ }
    private void disassembleEngine() { /*...*/ }
    private void repairBrokenParts() { /*...*/ }
    private void reassembleEngine() { /*...*/ }
}
```

Notice that the helper methods are **private** since users of this class probably do not need to call them. Here is an example showing how we might *attempt* to call these methods from some other class:

```
public class SomeCarApplicationProgram {
    public static void main(String[] args) {
        Car c = new Car();
        c.repair(); // OK to call this method
        c.disassembleEngine(); // Won't compile, since it is private
        c.repairBrokenParts(); // Won't compile, since it is private
    }
}
```

Now what about protecting an object's attributes? Well, the **public/private/protected** and *default* modifiers all work the same way as with behaviors. When used on instance variables, it allows others to be able to access/modify them according to the specified restrictions.

So far, we have never specified any modifiers for our attributes, allowing them all *default* access from classes within the same package or folder.

However, in real world situations, it is often best NOT to allow outside users to modify the internal private parts of your object. The reason is that results can often be disastrous. It is easy to relate to this because we well understand how we hide our own private parts ☺.




As an example, consider the following code, which may appear in any class. It shows that we can directly access the **balance** of a **BankAccount**.

This is clearly undesirable since there is little protection. Could you imagine if anyone could modify the balance of your bank account directly ?

```
BankAccount myAccount = new BankAccount("Mine");

myAccount.balance = 1000000.00f; // YAY

myAccount.balance = -1000000.00f; // WHY ...
```



In order to prevent direct access to important information we would need to prevent the code above from compiling/running. If we were to declare the **balance** instance variable as **private** within the **BankAccount** class, then the above code would not compile, thus solving the issue.

In general, while freedom to access/modify anything from anywhere seems like a friendly thing to do, it is certainly dangerous. Anyone could "stomp" all over our instance variables changing them at will. A general "rule-of-thumb" that should be followed is to declare ALL of your instance variables as **private** as follows:

```
public class Patient {
    private String name;
    private int age;
    private float height;
    private char gender;
    private boolean retired;

    ...
}
```

Once we do this, then the following code will not work (when written in a class other than the **Patient** class):

```
public class SomeApplicationProgram {
    public static void main(String[] args) {
        Patient p = new Patient();
        p.name = "Sandy Beach";           // will NOT compile
        p.age = 15;                       // will NOT compile
        p.height = 5.85f;                 // will NOT compile
        p.gender = 'M';                  // will NOT compile
        p.retired = false;                // will NOT compile
        System.out.println(p.name);      // will NOT compile
        System.out.println(p.age);       // will NOT compile
        System.out.println(p.height);    // will NOT compile
        System.out.println(p.gender);    // will NOT compile
        System.out.println(p.retired);   // will NOT compile
    }
}
```

What we have essentially done is to erect a wall around the object ... like the wall around a city. We have encapsulated it with a protective bubble. Although we are still able to create the object, we are prevented from accessing or modifying its internals now from outside the class. By doing this, we have protected the object so much that we cannot get information neither into it nor out from it. We have kind of secluded the object from the rest of the world by doing this. However, just as a walled city has gates or doors to allow access, we too have a form of gated access by means of any publicly available methods.



We will grant access to "some" of our object's attributes (i.e., instance variables) by creating methods known as **get** and **set** methods (also known as **getters** and **setters**). The idea of creating these gateways to our object's data is common practice and is considered to be a robust strategy when creating classes to be used in a large software application.

In this course, since we are only creating a few classes and since we are the only code writers, we may not immediately see the benefits of declaring **private** attributes and then creating these methods. However, in a larger/complicated system with hundreds of classes, the benefits become quite clear:

- object attributes are easier to understand and use
- attributes are protected from external/unknown changes
- we are following proper and robust coding style

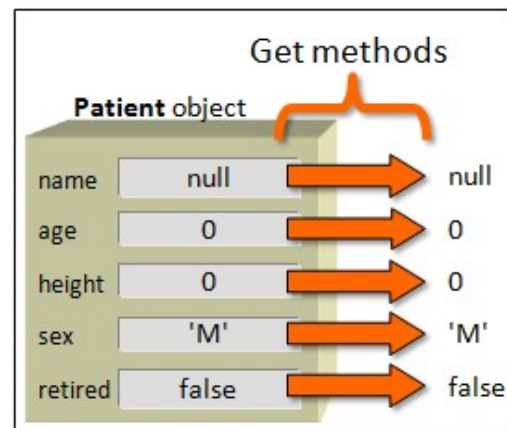
Let us first consider **get** methods. They let us look at information that is within the object by getting the object's attribute values (i.e., get the values of the instance variables). **Get** methods have:

- **public** access
- return type matching attribute's type
- name matching attribute's name
- code returning attribute's value

Here is how we would write the standard **get** methods for a **Patient** class:

```
public class Patient {
    private String name;
    private int age;
    private float height;
    private char gender;
    private boolean retired;

    // Get methods for name, age, height, gender and retired attributes
    public String getName() { return this.name; }
    public int getAge() { return this.age; }
    public float getHeight() { return this.height; }
    public char getGender() { return this.gender; }
    public boolean isRetired() { return this.retired; }
}
```



Notice that all the methods look the same in structure. They are all **public**, all have return types and names that match the attribute type, all have no parameters and all are one line long.

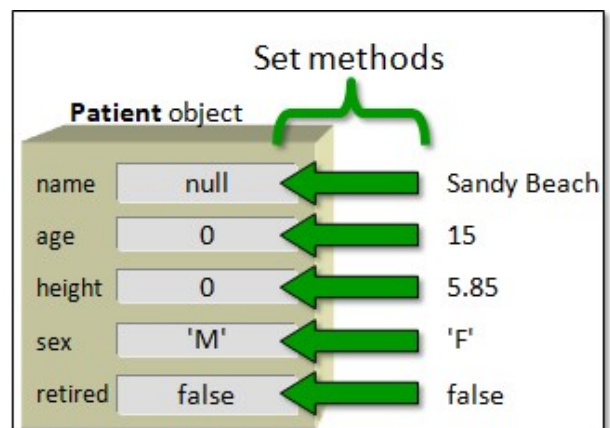
When we call the method to get the attribute value, the method simply returns the attribute value to us. It's quite simple. By convention, all get methods start with "**get**" followed by the attribute name, with the exception of attributes that are of type **boolean**. In that case, we usually use "**is**" followed by the attribute name, as it makes the method call more natural.

Now let us examine the **set** methods. **Set** methods allow us to put values into the instance variables (i.e., to set the object's attributes). **Set** methods have:

- **public** access
- **void** return type
- name matching attribute's name
- a parameter matching attribute's type
- code giving the attribute a value

Here is how we would write the standard **set** methods for the **Patient** class:

```
// Set method for name attribute
public void setName(String n) {
    this.name = n;
}
// Set method for age attribute
public void setAge(int a) {
    this.age = a;
}
// Set method for height attribute
public void setHeight(float h) {
    this.height = h;
}
// Set method for gender attribute
public void setGender(char g) {
    this.gender = g;
}
// Set method for retired attribute
public void setRetired(boolean r) {
    this.retired = r;
}
```



The single line of code in a **set** method is quite simple also.

When we call the method to give the attribute a new value (i.e., we supply the new value as a parameter to the method), the method simply takes that new attribute value and sets the attribute to it by using the **=** operator.

Normally, we write all the **get** and **set** methods together, and sometimes shorten them onto one line. Also, they are often listed in the code right after the **public** constructors as follows:

```

public class Patient {
    private String    name;
    private int       age;
    private float     height;
    private char      gender;
    private boolean   retired;

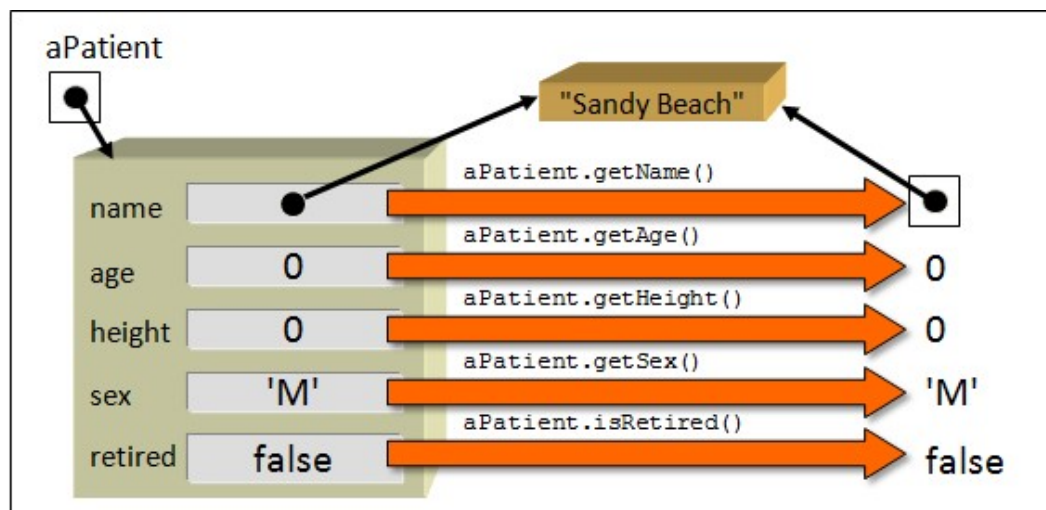
    // Constructor
    public Patient() {
        name = "Unknown";
        age = 0;
        height = 0;
        gender = '?';
        retired = false;
    }

    // Get methods
    public String getName() { return this.name; }
    public int getAge() { return this.age; }
    public float getHeight() { return this.height; }
    public char getGender() { return this.gender; }
    public boolean isRetired() { return this.retired; }

    // Set methods
    public void setName(String n) { this.name = n; }
    public void setAge(int a) { this.age = a; }
    public void setHeight(float h) { this.height = h; }
    public void setGender(char g) { this.gender = g; }
    public void setRetired(boolean r) { this.retired = r; }
}

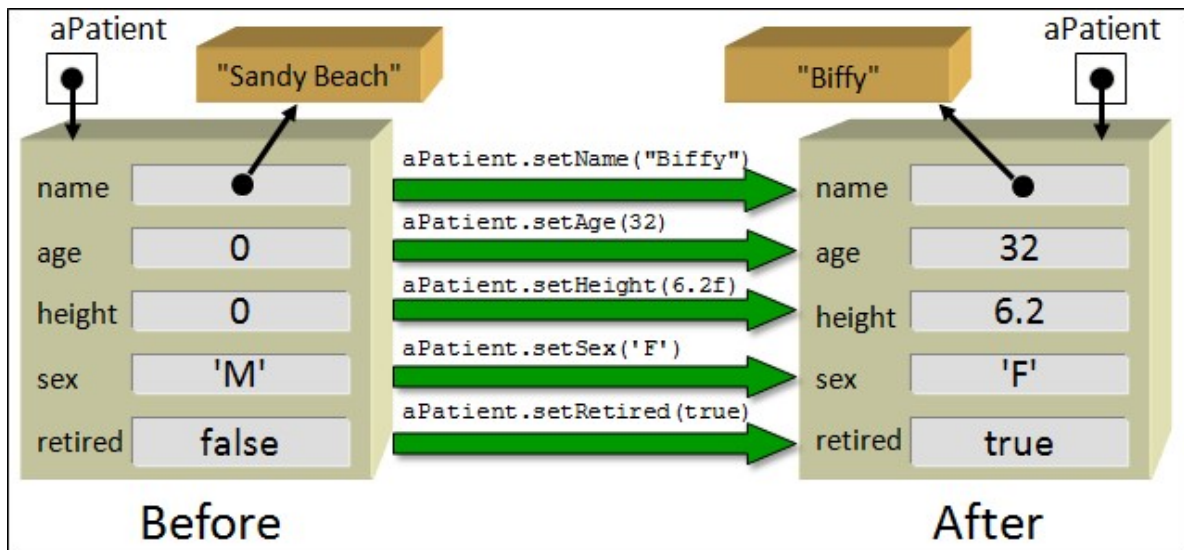
```

Here is how the **get** method works:



Notice that primitive attribute values are returned as simple values but *object* attribute values are returned as pointers/references to the object. The **Patient** object remains unchanged as a result of a `get` method call.

Here is how the **set** method works:



Notice that primitive attribute values are simply replaced with the new value. For object attribute values, after the **set** call, the attribute will point to the new object. The previous object that the attribute used to point to is discarded (i.e., garbage collected) if no other objects are holding on to it. Once we create these **get/set** methods, we can then access and modify the object from anywhere in our program as before:

```
public class TestPatientProgram {
    public static void main(String[] args) {
        Patient p = new Patient();

        System.out.println("Before Setting ...");
        System.out.println(p.getName()); // was println(p.name);
        System.out.println(p.getAge()); // was println(p.age);
        System.out.println(p.getHeight()); // was println(p.height);
        System.out.println(p.getGender()); // was println(p.gender);
        System.out.println(p.isRetired()); // was println(p.retired);

        p.setName("Sandy Beach"); // was p.name = "Sandy Beach";
        p.setAge(15); // was p.age = 15;
        p.setHeight(5.85f); // was p.height = 5.85f;
        p.setGender('F'); // was p.gender = 'F';
        p.setRetired(true); // was p.retired = true;

        System.out.println("\nAfter Setting ...");
        System.out.println(p.getName()); // was println(p.name);
        System.out.println(p.getAge()); // was println(p.age);
        System.out.println(p.getHeight()); // was println(p.height);
        System.out.println(p.getGender()); // was println(p.gender);
        System.out.println(p.isRetired()); // was println(p.retired);
    }
}
```


Here is what the output would be (however, initial values depend on the **Patient** constructor):

```
Before Setting ...
Unknown
0
0.0
?
false

After Setting ...
Sandy Beach
15
5.85
F
true
```

Now if we think for a moment ... what did we really do by making all the **get** and **set** methods ? Really, we wrote a lot of code (e.g., 5 **get** methods and 5 **set** methods for the **Patient** class) but did not gain anything new. The code does the same thing as before. In fact, the test code seems longer and perhaps slower (since we are calling a method to get/set the instance variables for us instead of accessing them directly). So why did we do this ? Let us review the advantages again:

1. First, **get/set** methods actually make life simpler for users of your class because the user does not have to understand the “guts” of the object being used. It allows them to treat the object as a “black box”. The user does not need to know about all the instance variables. Some are used to hold data that is temporary or private. You should only create public **get** methods for the instance variables that the user of the class would need to know about.
2. Second, it prevents the users of a class from directly modifying the object's internals. Recall, for example, that we should never be able to directly change the balance of our bank account without going through the proper transaction procedures such as depositing and withdrawal. Of course, if we always create **public get/set** methods for all our attributes, then we still would have no such protection. So, it is important to create **set** methods **only** for the attributes that you want the user of the class to be able to change directly. Therefore, you do not always need to make **set** methods.



3.7 Changing How Objects Look When Printed

As just described, properly-designed model classes will use *encapsulation* to hide any unnecessary information from those who will make use of those classes and to keep things simple. Sometimes however, it is desirable to be able to visually distinguish one object from another. For example, consider the following code:

```
public class MyObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Patient()); // a patient object
        System.out.println(new Patient()); // another patient object
    }
}
```

The result on the screen is as follows:

```
Patient@7d8a992f
Patient@164f1d0d
```

By default, JAVA displays all of the objects that you make in this manner, showing you the type of object (i.e., the class name) followed by something that represents the object's location in memory. This format for displaying objects is not very useful for debugging. If we had a dozen or so **Patient** objects displayed in this manner, we would not be able to "pick out" one that we may be looking for. It would be more advantageous if we had something a little more descriptive ... perhaps showing the patient's name.

What JAVA happens to be doing here is converting the **Patient** object to a **String** object first and then displaying the resulting characters to the screen. In fact, every object in JAVA has, by default, a method called **toString()** which will convert the object to a **String**.



The **Strings** returned from the call to **toString()** have the exact same characters that are displayed when we just display the objects directly using **System.out.println()**. That is because whenever JAVA attempts to display anything to the console, it automatically calls the **toString()** method for the object to convert it to characters before displaying. So, the two lines shown below do exactly the same thing:

```
Patient p = new Patient();

System.out.println(p); // displays Patient@7d8a992f
System.out.println(p.toString()); // displays Patient@7d8a992f
```

Why do we care ? Well, we can actually replace the default **toString()** behavior by writing our own **toString()** method for all of our own objects that defines exactly how to convert our object to a **String**. That is, we can control the way our object “looks” when we print it on the screen or when we display it in our user interface.

Suppose that we want our **Patient** object to display something like this when printed:

Patient named Hank

You should notice that the first two words of this output are “fixed” and it is only the last part (i.e., the first name of the **Patient**) that varies from patient to patient. We can make this to be the standard output format for all **Patient** objects simply by writing the following method in the **Patient** class:

```
public String toString() {  
    return ("Patient named " + this.name);  
}
```

This method overrides the default **toString()** method, essentially replacing it. Notice that the method is called **toString()** with no parameters and that it has a return type of **String**. This is important in order for the method to properly override the one inherited from the **Object** class.

Consider the output of the following code:

```
Patient          p1, p2, p3;  
  
p1 = new Patient();    // assume first name is set to "" within constructor  
p2 = new Patient();  
p2.setName("Holly");  
p3 = new Patient();  
p3.setName("Hank");  
  
System.out.println(p1);  
System.out.println(p2);  
System.out.println(p3);
```

Here is the output ...

```
Patient named  
Patient named Holly  
Patient named Hank
```

Now what if we wanted the output to be in this format instead:

19 year old Patient named Hank

To write an appropriate **toString()** method, we need to understand what is fixed in this output and what will vary. The number **19** should vary for each patient as well as the **first** and **last** names. Here is how we could write the code (replacing our previous **toString()** method):

```
public String toString() {  
    return (this.age + " year old Patient named " + this.name);  
}
```

Notice that the basic idea behind creating a **toString()** method is to simply keep joining together **String** pieces to form the resulting **String**. Now here is a harder one. Let us see if we can make it into this format:

19 year old non-retired patient named Hank

Here we have the **age** and **names** being variable again but now we also have the added variance of their *retirement* status.

Here is one attempt:

```
public String toString() {  
    return (this.age + " year old " + this.retired + " patient named "  
        + this.name);  
}
```

However, this is not quite correct. This would be the format we would end up with:

19 year old false patient named Hank

Notice that we cannot simply display the value of the **retired** attribute but instead need to write “retired” or “non-retired” for the **retired** status.

To do this then, we will need to use an **IF** statement. However, in JAVA, we cannot write an **IF** statement in the middle of a **return** statement. So we will need to do this using more than one line of code. We can make an **answer** variable to hold the result and then break down our method into logical pieces that append to this **answer**:

```
public String toString() {  
    String answer;  
  
    answer = this.age + " year old ";  
    answer = answer + this.retired;  
    answer = answer + " patient named " + this.name);  
  
    return answer;  
}
```

Now we can insert the appropriate **IF** statements as follows:

```
public String toString() {  
    String  answer;  
  
    answer = this.age + " year old ";  
  
    if (this.retired)  
        answer = answer + "retired";  
    else  
        answer = answer + "non-retired";  
    answer = answer + " patient named " + this.name;  
  
    return answer;  
}
```

The result is what we wanted. Note however, that we can simplify this code a little further:

```
public String toString() {  
    String  answer = this.age + " year old ";  
  
    if (!this.retired)  
        answer = answer + "non-";  
  
    return (answer + "retired patient named " + this.name);  
}
```

3.8 A Bank Example

Consider implementing some software for a bank. Likely we need a **Bank** object that will contain **BankAccount** objects where each account is owned by a bank **Customer**. So, we will need a few interacting objects. Let's begin with a **Customer** object. We can define it as a simpler version to a **Person** object with get/set methods and a **toString()** method as follows:

```
public class Customer {
    private String    firstName;
    private String    lastName;
    private Address    address;
    private String    phoneNumber;

    // This is the zero-parameter constructor
    public Customer() {
        firstName = "UNKNOWN";
        lastName = "UNKNOWN";
        address = null;
        phoneNumber = "(???) ???-????";
    }

    // This is a 4-parameter constructor
    public Customer (String f, String l, Address a, String p) {
        firstName = f;
        lastName = l;
        address = a;
        phoneNumber = p;
    }

    // These are the get/set methods
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public Address getAddress() { return address; }
    public String getPhoneNumber() { return phoneNumber; }
    public void setFirstName(String s) { firstName = s; }
    public void setLastName(String s) { lastName = s; }
    public void setAddress(Address a) { address = a; }
    public void setPhoneNumber(String p) { phoneNumber = p; }

    // This returns a String representation of the customer
    public String toString() {
        return "Customer: " + firstName + " " + lastName +
            " living at " + address;
    }
}
```

Of course, we will need to make the **Address** object too. We can make something quite simple like this (we will leave off city/province/postal code to keep things simple):

```

public class Address {
    private String    streetNumber;
    private String    streetName;

    // This is the 2-parameter constructor
    public Address(String number, String name) {
        streetNumber = number;
        streetName = name;
    }

    // These are the get/set methods
    public String getStreetNumber() { return streetNumber; }
    public String getStreetName() { return streetName; }
    public void setStreetName(String s) { streetName = s; }
    public void setStreetNumber(String s) { streetNumber = s; }

    // This returns a String representation of the address
    public String toString() {
        return streetNumber + " " + streetName;
    }
}

```

Now, let us define a **BankAccount** with the following attributes and constructors as follows:

```

public class BankAccount {
    private Customer  owner;
    private int       accountNumber;
    private float     balance;
}

```

Likely, when someone makes a new **BankAccount**, they DO NOT get to choose their own account number, as this is usually assigned by the bank itself. Let us assume that the first created account is assigned the account number 100001, the second gets 100002, the third 100003 and so on. In this scenario, we can simply keep a counter that starts at 100001 and increases each time a new account is created.

To do this, we can create a static/class variable in the **BankAccount** class to represent this counter. We can call it **LAST_ACCOUNT_NUMBER** which will store the account number that was last given out. We can give this variable an initial value of 100000 as follows ...

```

private static int LAST_ACCOUNT_NUMBER = 100000;

```

Then, when a new **BankAccount** is created, we can give it an **accountNumber** which is one more than the **LAST_ACCOUNT_NUMBER** and then increment this counter to get it ready for the next time. This counter of ours will work exactly like one of those ticket dispensers when you wait in line at a store.

This can be done by adjusting all of the **BankAccount** constructors so that they do not allow the user to "specify" the **accountNumber**. But



rather set it to the next available number and then increment the counter. Here is the code that we would need to write:

```
public class BankAccount {
    private static int LAST_ACCOUNT_NUMBER = 100000;

    private Customer    owner;
    private int         accountNumber;
    private float       balance;

    // This is the zero-parameter constructor
    public BankAccount() {
        owner = null;
        accountNumber = ++LAST_ACCOUNT_NUMBER;
        balance = 0;
    }

    // This is a 1-parameter constructor
    public BankAccount(Customer c) {
        owner = c;
        balance = 0;
        accountNumber = ++LAST_ACCOUNT_NUMBER;
    }

    // These are the get methods
    public Customer getOwner() { return owner; }
    public int     getAccountNumber() { return accountNumber; }
    public float   getBalance() { return balance; }
}
```

Notice that each bank account will always get a new number because all available constructors increment the global counter before assigning the bank account number to the new account. Also, notice that there are no **set** methods. That is because an account should never be allowed to change its owner nor its accountNumber once it has been created. Also, there should not be any permission to directly modify (i.e., set) the balance ... there should be **deposit** and **withdrawal** procedures that must be followed.

Now, what about a **toString()** method ? What should a bank account look like when printed ? That is up to us. Perhaps we want it to look like this:

Bank Account #100001 with balance \$1765.92

The above does not display the account owner. Here is how we would write the code:

```
public String toString() {
    return "Bank Account #" + accountNumber + " with balance $" +
        String.format("%,1.2f", balance);
}
```

Of course, we need a way of depositing and withdrawing money:

```
public void deposit(float amount) {  
    balance += amount;  
}
```

```
public boolean withdraw(float amount) {  
    if (amount <= balance) {  
        balance -= amount;  
        return true;  
    }  
    return false;  
}
```

Notice that the **withdraw()** method returns a **boolean** that will inform us as to whether or not there was enough money in the account. Both of these methods would need to be added to the account.

Here is a test program to see if it all works:

```
public class AccountTestProgram {  
    public static void main(String args[]) {  
        BankAccount    b1, b2, b3;  
  
        b1 = new BankAccount(new Customer("Tim", "Foil",  
                                           new Address("12", "Elm St.", "613-555-5555"));  
        b2 = new BankAccount(new Customer("Dan", "Sing",  
                                           new Address("1267A", "Oak St.", "613-555-5556"));  
        b3 = new BankAccount(new Customer("Fran", "Tick",  
                                           new Address("4761", "Pine Cres.", "613-555-5557"));  
  
        b1.deposit(125);  
        b2.deposit(3245.02f);  
        b2.withdraw(1000);  
        b3.withdraw(20);  
  
        System.out.println(b1);  
        System.out.println(b2);  
        System.out.println(b3);  
    }  
}
```

Here is the expected output:

```
Bank Account #100001 with balance $125.00  
Bank Account #100002 with balance $2,245.02  
Bank Account #100003 with balance $0.00
```

Notice that the account numbers assigned are consecutive (i.e., 100001, 100002 and 100003). Of course, each time we run the program, the account numbers start over at 100001 again. If we wanted to ensure that our code assigned new numbers even when we restart the program, we would have to store the last account number counter in a file and then re-save the changed counter each time to the file. We will discuss the reading and writing of files later in the course.

Finally, we need a way of keeping the accounts all together. We can do this by making a **Bank** class which keeps an array of **BankAccount** objects. Since we are using arrays, we will also want to define a fixed size for the array ... perhaps defined as a constant. Here is what we can do:

```
public class Bank {
    private static final int    ACCOUNT_CAPACITY = 100;

    private String              name;
    private BankAccount[]      accounts;
    private int                 numberOfAccounts;

    public Bank(String n) {
        name = n;
        numberOfAccounts = 0;
        accounts = new BankAccount[ACCOUNT_CAPACITY];
    }

    // These are the get methods (set methods are not allowed)
    public String getName() { return name; }
    public BankAccount[] getAccounts() { return accounts; }
    public int getNumberOfAccounts() { return numberOfAccounts; }

    // This returns a string representation of the bank
    public String toString() {
        return name + " with " + numberOfAccounts + " accounts";
    }
}
```

Of course, we need a way of opening accounts at the bank:

```
// Add an account to the bank
private void addAccount(BankAccount b){
    if (numberOfAccounts < ACCOUNT_CAPACITY)
        accounts[numberOfAccounts++] = b;
}
```

Notice that the method is **private**. That is because we don't want others passing in accounts that may be invalid or ones that belong to different banks. Instead, we will make a **public** method as a means of creating a new account, given a **Customer**:

```
// Open a bank account for this customer
public void openAccount(Customer c){
    addAccount(new BankAccount(c));
}
```

We will want to also probably allow depositing and withdrawals from the accounts based on an account number:

```
// Deposit an amount of money into account with given accountNumber
public boolean deposit(int accNum, float amount) {
    for (int i=0; i<numberOfAccounts; i++) {
        if (accounts[i].getAccountNumber() == accNum) {
            accounts[i].deposit(amount);
            return true;
        }
    }
    return false;
}
```

```
// Withdraw an amount of money from account with given accountNumber
public boolean withdraw(int accNum, float amount) {
    for (int i=0; i<numberOfAccounts; i++) {
        if (accounts[i].getAccountNumber() == accNum)
            return accounts[i].withdraw(amount);
    }
    return false;
}
```

We can then write any interesting methods that we want such as these:

```
// Determine total of all account balances
public float totalOfAllBalances() {
    float answer = 0;

    for (int i=0; i<numberOfAccounts; i++) {
        answer += accounts[i].getBalance();
    }
    return answer;
}
```

```
// List all accounts
public void listAccounts() {
    for (int i=0; i<numberOfAccounts; i++)
        System.out.println(accounts[i]);
}
```

We can test it all out using the following program:

```
public class BankTestProgram {
    public static void main(String[] args) {
        // Make a Bank
        Bank myBank = new Bank("Mark's Bank");

        // Make some bank accounts with customers
        myBank.openAccount(new Customer("Tim", "Foil",
            new Address("12", "Elm St.", "613-555-5555"));
        myBank.openAccount(new Customer("Dan", "Sing",
            new Address("1267A", "Oak St.", "613-555-5556"));
        myBank.openAccount(new Customer("Fran", "Tick",
            new Address("4761", "Pine Cres.", "613-555-5557"));

        myBank.deposit(100001, 125);
        myBank.deposit(100002, 3245.02f);
        myBank.withdraw(100002, 1000);
        myBank.withdraw(100003, 20);

        System.out.println("\nHere are the bank accounts:");
        myBank.listAccounts();

        System.out.println("\n\nThe bank has this much money: $" +
            String.format("%.1.2f", myBank.totalOfAllBalances()));
    }
}
```

Here is the expected output:

```
Here are the bank accounts:
Bank Account #100001 with balance $125.00
Bank Account #100002 with balance $2,245.02
Bank Account #100003 with balance $0.00
```

```
The bank has this much money: $2,370.02
```

As you can see, object-oriented programming requires you to define and implement many objects and to get them to work together in meaningful ways. Often, the object class definitions that you write can be re-used in many applications. It is therefore a good idea to ensure that these objects are robust and that their methods provide proper results. To do this, you should perform proper testing of your objects.



Unfortunately, testing is often tedious. It is therefore poorly done and ignored by many programmers. Companies that hire programmers do not like laziness ... and even worse ... they hate code with bugs or errors in it. To avoid disappointing your boss, possibly losing your job, and just to feel good about the quality of your work ... you should properly test your code.

Normally, it is not common to test your constructors nor get/set methods, but it is certainly important to test methods that perform computations, search, sort, etc... For problems that require numerical parameters, it is a good idea to test different values that could potentially cause problems. For example, if we were to fully test the **deposit()** method for the **BankAccount** class, we would want to test depositing the following amounts:

- 0.0 // deposit nothing
- 0.67 // a cents amount
- 100.57 // a typical positive amount
- 100.2234343 // an amount with many decimal places
- -34 // an invalid amount

We could create a simple test program to do this, making sure that we properly display the results to confirm that they are correct as follows ...

```
public class DepositTestProgram {
    public static void main(String args[]) {
        BankAccount acc;

        acc = new BankAccount(new Customer("Rusty", "Can",
                                           new Address("33", "Birch Ave."), "613-555-5558"));
        System.out.println("Account at start: " + acc);
        acc.deposit(0.0f);
        System.out.println("Account after depositing $0.00: " + acc);
        acc.deposit(0.67f);
        System.out.println("Account after depositing $0.67: " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57: " + acc);
        acc.deposit(100.2234343f);
        System.out.println("Account after depositing $100.2234343: " + acc);
        acc.deposit(-34);
        System.out.println("Account after depositing $-34: " + acc);
    }
}
```

Here is the output:

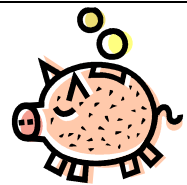
```
Account at start:                Account #100000 with $0.00
Account after depositing $0.00:  Account #100000 with $0.00
Account after depositing $0.67:  Account #100000 with $0.67
Account after depositing $100.57: Account #100000 with $101.24
Account after depositing $100.2234343: Account #100000 with $201.46
Account after depositing $-34:   Account #100000 with $167.46
```

Notice the careful use of **System.out.println()** in the program to provide a kind of “log” showing exactly what we tested and the order that things were tested in. If you were to read the output, you should be able to follow along as the deposit transactions were made to confirm the correct balance each time.

From the output, you may notice something that needs changing. For example, you may decide to prevent depositing negative amounts of money. You might do this by changing the code to generate an exception (more on this later) or perhaps simply perform a check and ignore deposits of negative amounts.

It really depends on the application and whether or not it is tied-in with the user interface. For example, at a bank machine, it is impossible to deposit a negative amount of money because the machine does not allow you to enter a negative sign. In such a situation, you may choose simply to ignore the problem altogether, since it would never occur. However, a simple check may be best, in case you port your code into a different program:

```
public void deposit(float amount) {
    if (amount > 0)
        balance += amount;
}
```



Then we would re-run the same test code to see whether or not it worked:

```
Account at start:                Account #100000 with $0.00
Account after depositing $0.00:  Account #100000 with $0.00
Account after depositing $0.67:  Account #100000 with $0.67
Account after depositing $100.57: Account #100000 with $101.24
Account after depositing $100.2234343: Account #100000 with $201.46
Account after depositing $-34:   Account #100000 with $201.46
```

Now this was a simple test program which is often known as a “Test Unit”. In larger, more complicated, real-world programs, in order to keep organized, it would be necessary to create multiple simple test units that test particular aspects of the program. For example,

```

public class BankAccountTestUnit1 {
    public static void main(String args[]) {
        BankAccount acc = new BankAccount(new Customer("Rusty", "Can",
            new Address("33", "Birch Ave."), "613-555-5558"));

        System.out.println("Account before depositing $100.57: " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57: " + acc);
    }
}

```

```

public class BankAccountTestUnit2 {
    public static void main(String args[]) {
        BankAccount acc = new BankAccount(new Customer("Rusty", "Can",
            new Address("33", "Birch Ave."), "613-555-5558"));

        System.out.println("Account before withdrawing $100: " + acc);
        acc.withdraw(100);
        System.out.println("Account after withdrawing $100: " + acc);
    }
}

```

In fact, it is often the case that we would like to perform transactions and test cases on a particular bank account. In this case, we can break down the separate test units as test methods in a larger test program:

```

public class BankAccountTestUnit3 {
    public static void deposit1(BankAccount acc) {
        System.out.println("Account before depositing $100.57: " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57: " + acc);
    }
    public static void deposit2(BankAccount acc) {
        System.out.println("Account before depositing $0.01: " + acc);
        acc.deposit(0.01f);
        System.out.println("Account after depositing $0.01: " + acc);
    }
    public static void withdraw1(BankAccount acc) {
        System.out.println("Account before withdrawing $100.57: " + acc);
        acc.withdraw(100.57f);
        System.out.println("Account after withdrawing $100.57: " + acc);
    }
    public static void withdraw2(BankAccount acc) {
        System.out.println("Account before withdrawing $0.01: " + acc);
        acc.withdraw(0.01f);
        System.out.println("Account after withdrawing $0.01: " + acc);
    }
}

```



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

    acc = new BankAccount(new Customer("Rusty", "Can",
                                       new Address("33", "Birch Ave."), "613-555-5558"));
    acc.deposit(0);

    deposit1(acc);
    deposit2(acc);
    withdraw1(acc);
    withdraw2(acc);

    acc = new BankAccount(new Customer("Ann", "Tenna",
                                       new Address("84", "Maple Ave."), "613-555-5559"));
    acc.deposit(10);

    deposit1(acc);
    deposit2(acc);
    withdraw1(acc);
    withdraw2(acc);

    acc = new BankAccount(new Customer("Ella", "Vator",
                                       new Address("873", "Spruce Dr."), "613-555-5560"));
    acc.deposit(200);

    deposit1(acc);
    deposit2(acc);
    withdraw1(acc);
    withdraw2(acc);
}
```

There are actually principles and guidelines for writing test cases for large systems. However, it is beyond the scope of this course. You will learn more about proper testing next year.