

---

## Chapter 4

# Defining Your Own Functions/Methods

---

## What is in This Chapter ?

Object-Oriented Programming (OOP) involves creating objects of our own. In this set of notes, we will discuss how to write functions and procedures for our objects, which are also called **methods**. Methods are the set of instructions that perform some operations with the data or objects that you need to work with. You will spend 99% of your time writing methods in JAVA, since all code must belong to some method. Finally, we will discuss **static** methods, which are more general methods that do not need an object in order to compute something.



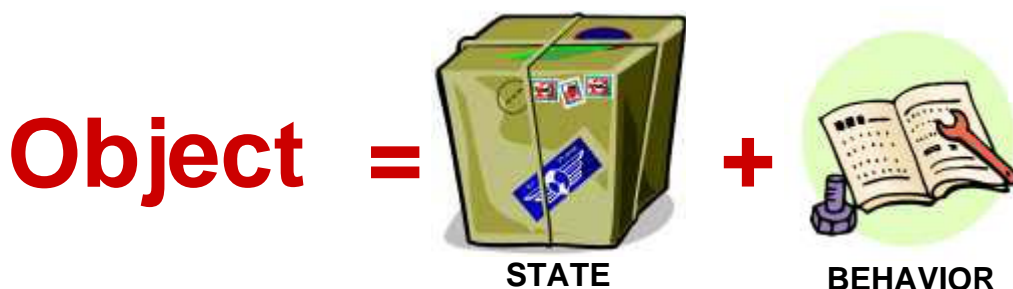
## 4.1 What are Methods ?

At this point in the course, we have already defined a few of our own objects (i.e., **Car**, **Person**, **Address**, **BankAccount**, etc.). Do you remember why we defined them ? We did it so that we could group information together in a meaningful way. For example, we associated information such as **firstName**, **lastName**, **age**, **gender** and **retired** as all being part of a **Person** object. This idea of grouping information/attributes together relates to real-life since all real life objects (tangible or not) are defined by their attributes. Over time, these attributes will change. Hence at any time in the object's existence, the combined attributes represent the object's **current state** at that point in time.

There is also something else that real objects have in addition to their state ... they have **behavior**. That is, objects in general, can **do** things and when used, they respond in certain ways. For example, you can ask a person "What is your name ?", and they usually respond by stating the value of their **name** attribute. Also, a person can be asked to "Stand up!", and we usually expect that the person would change their position from a lying/sitting position to a standing position.

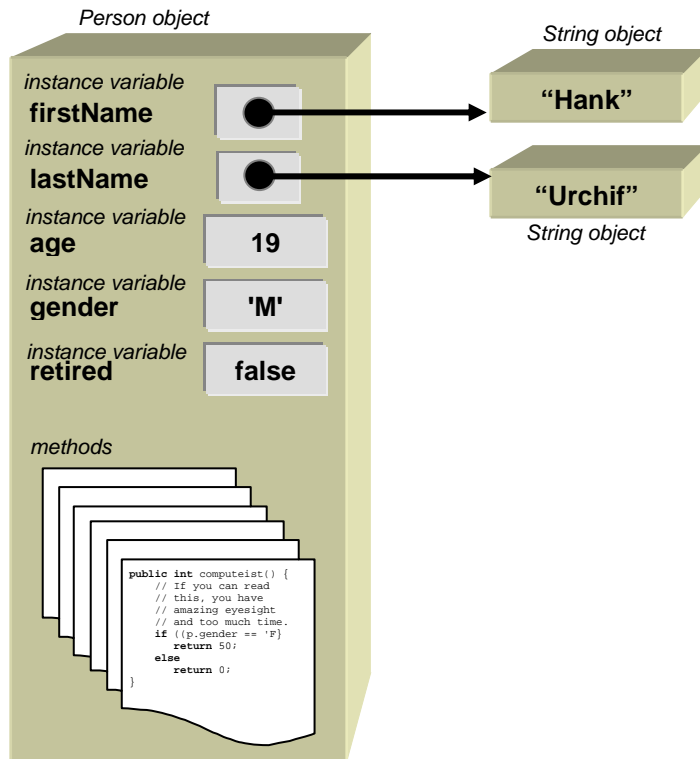
Objects also have the ability to change their own state as well as the state of other objects. For example, if a **Person** spends money from their bank account, the account's balance is changed afterwards. Also, if a **Person** gives money to another **Person**, one person's money increases, while the other's money decreases.

Object-Oriented programming languages such as JAVA, allow us to define both state and behavior for our objects. In fact, objects will generally have a whole set of pre-defined functions/behaviors that we can use to modify and manipulate the object in some way. It is important that you understand that Objects are defined by two things: (1) their state, and (2) their behaviors:



Since we have already discussed how to define state (or attributes, instance variables) for our objects, it is now time to discuss how we can define the object's behaviors. All object behaviors are defined individually, one at a time. Each behavior is defined in something called a **method**. In English, the word "method" is defined as "a way of doing something". So, we will actually be defining "how" the object will behave (i.e., what it needs to do) when it is asked to exhibit one of its behaviors.

So, you can think of each object in JAVA as having a list of instance variables as well as a list of method definitions:



There are basically two types of methods in JAVA:

- Functions – methods that do something and then return a value
- Procedures – methods that do something but do not return a value.

## 4.2 Defining Methods - Functions

To help motivate our discussion, recall this code that we used to determine a discount for “Grandma/Granddaughter Night” at the theatre. Recall that the discount should be 50% for women that are retired or to girls who are 12 and under. For all other people, the discount should otherwise be 0%:

```
Person    p = new Person();

... // some code omitted to set the name, age, gender, etc...

// Write code to compute the appropriate discount
int discount = 0;

if ((p.gender == 'F') && (p.age < 13 || p.retired))
    discount = 50;
```

Recall that the code `p.age` actually goes into the `p` `Person` object and retrieves its `age` attribute's value. Similarly, the `p.gender` code retrieves the `gender` value from `Person p`. We can actually define a function in the `Person` class that computes the appropriate discount for the person. Such a function should return an integer answer of either 0 or 50, representing the percentage of discount that the `Person` is entitled to have.

To do this, we need to understand first how the function is to be used. In JAVA, all functions/methods are invoked (i.e., called or used) by using the dot `.` operator on the object whose method you want to use. We then need to have a meaningful name for the method. In our example, perhaps we can call it `computeDiscount` since it aptly describes what we are trying to do. Method calls all require round brackets `()` at the end. So to *call* the method, we may write something like this:

```
p.computeDiscount()
```

Since this function will return the discount value, we can then make use of the value returned from this method call and simplify our code as follows:

```
Person    p = new Person();

... // some code omitted to set the name, age, gender, etc...

// Write code to compute the appropriate discount
int    discount = p.computeDiscount();
```

Of course though, the code above does not compile, at least not until we *define* the method called `computeDiscount` in the `Person` class. So now lets go do that.

Recall that the `Person` class already has instance variables defined in it as well as one or more constructors. To define a new behavior for the `Person`, we simply add a new method ... usually at the bottom of the class but still within the final closing brace `}` for the class ... as shown here:

So how do we write the method? The steps are given on the next page.

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

    // These are the constructors
    Person() {
        ...
    }
    Person(String fn, String ln, int a, char g, boolean r) {
        ...
    }
    // Start writing your methods here
    ...
}
```

Put your methods here

## STEP 1 – METHOD SIGNATURE:

The 1<sup>st</sup> step is to write the method's signature. The method signature is the first line of the method where you specify its name, parameters and return type. It is called a signature because it is what makes the method unique with respect to any other methods in this class.

Here is the basic format of our method:

```
int computeDiscount() {  
}
```

Notice that the first word in the method's signature is the **return type** of the method (i.e., the type of thing (i.e., primitive or object) that will be returned by the method). In our example, we are trying to compute a discount value which is an integer. Hence, we specify **int** to be the type of thing returned. The next part of the signature is the **method name**. Just like variable names, the method name should have no spaces or weird characters and should be lower case with capitalized words (except the first). The method name should be immediately followed by opening and closing round brackets **()**. Finally, we use the braces **{ }** to encapsulate the code that is to be evaluated when the method is called ... the code within the braces is called the **method body** ... which is currently empty at the moment.

## STEP 2 – RETURN VALUE:

The 2<sup>nd</sup> step when writing a method is to create a temporary variable of the same type as the return type and to return it at the end of the method. (This step is not required, but it always helps you “keep your head straight” when you are learning to write methods. After you complete this step, your method should compile fine, although it is not complete). Here is how we can do this for our example:

```
int computeDiscount() {  
    int answer = 0;  
  
    return answer;  
}
```

Notice that the variable `answer` will contain the final answer for our method. We start off by assuming a value of 0 but will modify this soon. The last line of the method returns the variable's value. In JAVA, the **return** statement is used to stop the method immediately and return with the specified value.

### **STEP 3 –METHOD BODY:**

The 3<sup>rd</sup> and last step is to write the remaining part of the method body. This is the interesting part that requires you to think. In our example, we need to write code that will determine the correct discount. How can we do this? Well, we actually already did this previously so we can start with our previous code. Here is how we can do this, although some changes are needed:

```
int computeDiscount() {
    int    answer = 0;

    int    discount = 0;
    if ((p.gender == 'F') && (p.age < 13 || p.retired))
        discount = 50;

    return answer;
}
```

In the code above, you will notice some issues. To begin, we can replace the **discount** variable with our new **answer** variable which will contain the proper discount at the end of the method.

```
int computeDiscount() {
    int answer = 0;

    if ((p.gender == 'F') && (p.age < 13 || p.retired))
        answer = 50;

    return answer;
}
```

Next, you will notice that the variable **p** is not defined anywhere in this method. This variable was the variable defined in our other program and it was used to represent the person for whom we are to compute the discount. **p.gender** actually went into **Person** object **p** and retrieved that person's gender. Similarly **p.age** and **p.retired** retrieved those attributes of **p**. But we are now writing code that is *inside* the **Person** class definition. The code we are writing must work for all **Person** objects, not just **p**. Hence, the **p** must vary according to the person that we are trying to compute the discount for.

We need to replace **p** by the word **this**, because **this** is the placeholder (i.e., nickname) of the object for which we called this **computeDiscount** method. That is, whenever we are defining an object's behavior (i.e., writing a method), the word **this** represents the object that will be exhibiting the behavior ... the object whose data we need to access.

(For a review of the word **this**, perhaps go back and read the notes on constructors).

So, here is resulting code ...

```

int computeDiscount() {
    int answer = 0;

    if ((this.gender == 'F') && (this.age < 13 || this.retired))
        answer = 50;

    return answer;
}

```

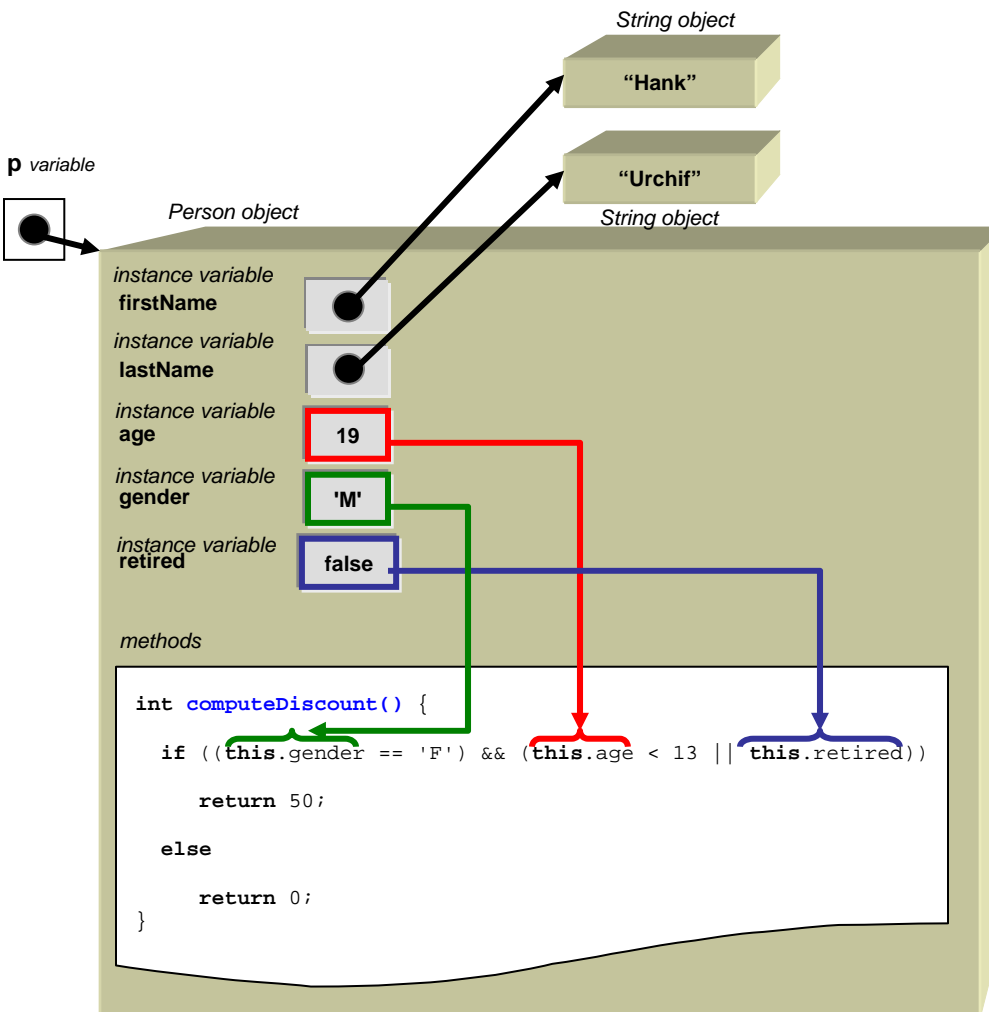
In fact, we can simplify this code even further by realizing that the method is simple enough to not need the **answer** variable:

```

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

```

Here is a picture of what is happening inside the object when we test it using the code `p.computeDiscount()`;



We have successfully defined our first method/behavior ! But ... practice makes perfect ... so let us do another method. Recall the code we wrote that determined the oldest **Person**:

```

Person    p1, p2, p3;

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

// Write code to set the oldest variable to be the oldest person.
Person    oldest;

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

```

Lets write a method called **isOlderThan()** which allows us to determine whether one **Person** is older than another. The method should return a **boolean** (i.e., true or false):

```

boolean isOlderThan() {
    boolean answer = false;
    ...
    return answer;
}

```

To **call** this method, we may write something like this: `p1.isOlderThan()`

But wait! The something seems to be missing. We need to know the *other* person that we want to compare **p1**'s age with. If we look at our original code, we are comparing **p1** with **p2**, **p1** with **p3**, **p2** with **p1**, and **p2** with **p3**. So we need a way of "telling the method" which person to compare **p1** with. This is additional information that the method requires.

Recall from our discussion on constructors that we can **pass-in** additional information as parameters between the round **()** brackets. So, we can pass in the person to compare with as follows: `p1.isOlderThan(p2)`

We then need to go into our method definition and define the incoming parameter by specifying its type as well as giving it a name:

```

boolean isOlderThan(Person x) {
    boolean answer = false;
    ...
    return answer;
}

```



Now the method has the two **Person** objects that are needed to make the comparison (i.e., **this** and **x**). So, how do we write the rest of the code? We just need to compare **this**'s age with **x**'s age as follows:

```
boolean isOlderThan(Person x) {
    boolean answer = false;

    if (this.age > x.age)
        answer = true;
    else
        answer = false;

    return answer;
}
```



And we are done! Remember though, that this can be simplified:

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

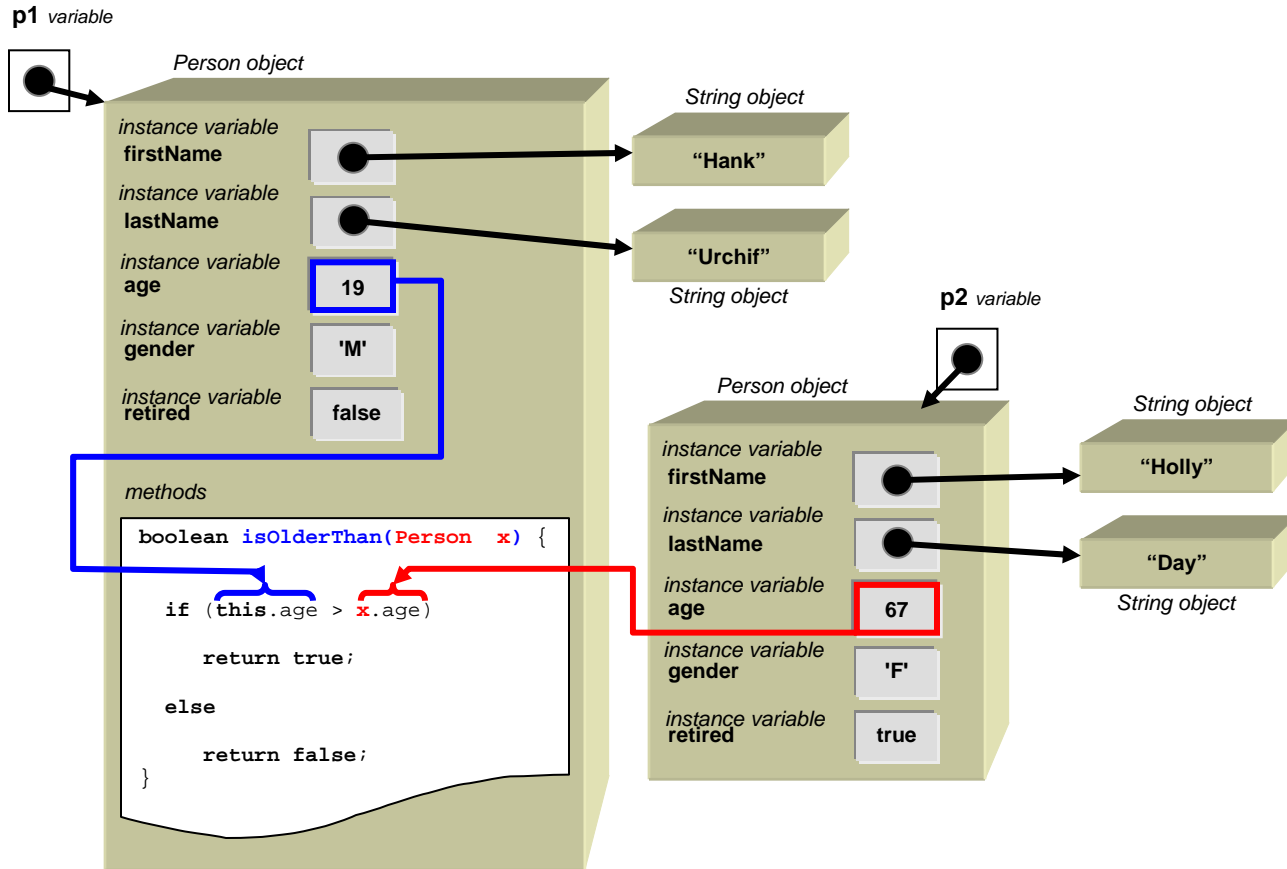
and even more if we really want to be efficient...

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

Here is what the resulting test code looks like when we make use of this method now:

```
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)`:



For an interesting exercise, how would we write another method so that it takes two `Person` objects as parameters and checks both of them? We would use it as follows:

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

Here is the new method (notice that it now takes 2 parameters):

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



You may have noticed that we wrote two methods with the same name (i.e., **isOlderThan**). How can JAVA allow this? That is, when we call the method, how does JAVA know which method to use?

Recall that the first line of the method is the method's **signature**. The signature is what makes it unique. In our first **isOlderThan()** method, there was one parameter. In the second **isOlderThan()** method there are two parameters. That is how JAVA distinguishes between the two methods. When calling a method, JAVA always looks at the method's name as well as its list of parameter types to determine which one to call.

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:

```
void eat(Apple x) { ... }
void eat(Orange x) { ... }
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 **anObject** 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 1<sup>st</sup> **eat()** method. If it is an **Orange** object, it will call the 2<sup>nd</sup> method. What if **z** is a Banana? It will NOT call the 3<sup>rd</sup> method ... because the 3<sup>rd</sup> method requires 2 Bananas and we are only passing in one. A call of **p.eat(z, z)** would call the 3<sup>rd</sup> 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:

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

You will get an error saying:

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

Getting back to writing methods, how can we write a method called `oldest()` that returns the oldest of the 3 `Person` objects? If we had such a method, then we can simplify our earlier test code to this one simple line:

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

Do you notice something different with the *type of thing* returned from the method? This method would now need to return a `Person` object (i.e., the oldest of the three) instead of a `boolean`. Hence, the return type for the method must be different. Here is what we would start with:

```
Person oldest(Person x, Person y) {
    Person answer;

    ...

    return answer;
}
```

Then, we simply check all of the ages as before:

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

You should notice that this code looks VERY much like our original test code. Basically, all we have done is moved the age-checking code into the method. So, a method is really just a *chunk of code that is defined for an object*.

## 4.3 Defining Methods - Procedures

Now let's write a couple of procedures (i.e., methods that do not return anything). Let's write a method called `retire()` that changes a `Person` object's `retired` state from `false` to `true`. You should realize that there is no "answer" for such a method. We simply need to change the value of the `retired` variable, but we don't need to return any specific object or primitive from

the method. In this case, there is no return value. In JAVA, when we do not have a value to return, we specify the return type for the method to be **void** as follows:

```
void retire() {
    ...
}
```

So what do we need to do in the method ? We just need to change the **retired** variable to **true**:

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

That's it! Lets try one more. How would we write a method called **swapNameWith()** that allows one **person** to change names with another **Person** ? Hopefully, you now understand that the return type should be **void**, since it performs an operation on the Person but no answer is required to be returned. Also, you should realize that the 2<sup>nd</sup> person should be passed-in as a parameter. Here is the structure:

```
void swapNameWith(Person x) {
    ...
}
```

How do we swap the names ? Well, the first person's **firstName** attribute should become the second person's **firstName** attribute and vice versa. We also need to do this for the **lastName** attributes. Here is a first attempt:

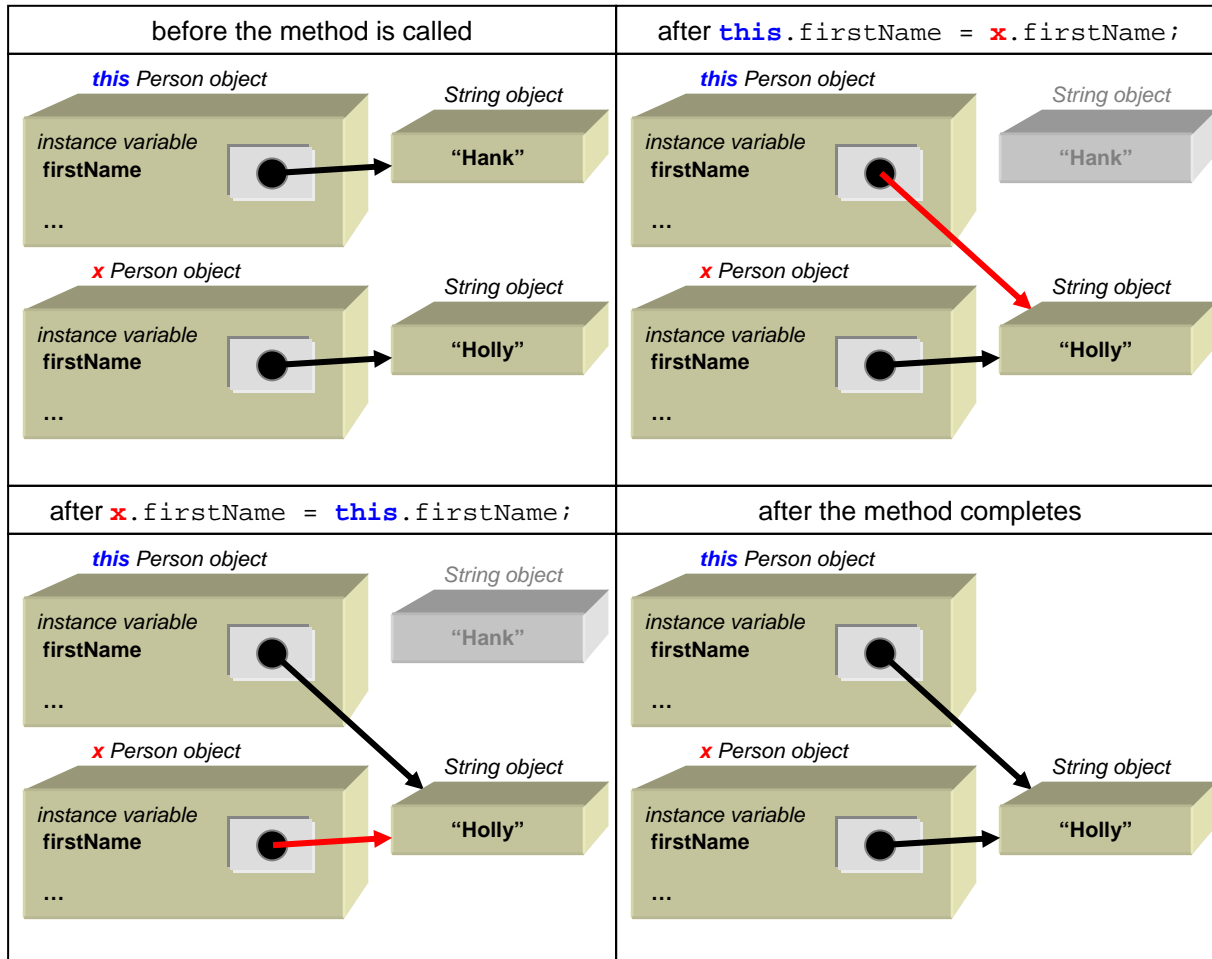
```
void swapNameWith(Person x) {
    // Swap the first names
    this.firstName = x.firstName;
    x.firstName = this.firstName;

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



But this code will not work. Why ? Well, the first line of code replaces **this**'s **firstName** with **x**'s **firstName**. Hence, we are overwriting (i.e., erasing) **this**'s **firstName** value ... and the value will be gone forever. So on the second line of code, when we try to put **this**'s **firstName** into **x**, we are actually putting in **x**'s **firstName** that we stored there from the first line. So both

persons will end up with the same **firstName**, which was **x**'s original **firstName**. The same problem occurs for the **lastName**. The following shows what happens with the **firstName** ...



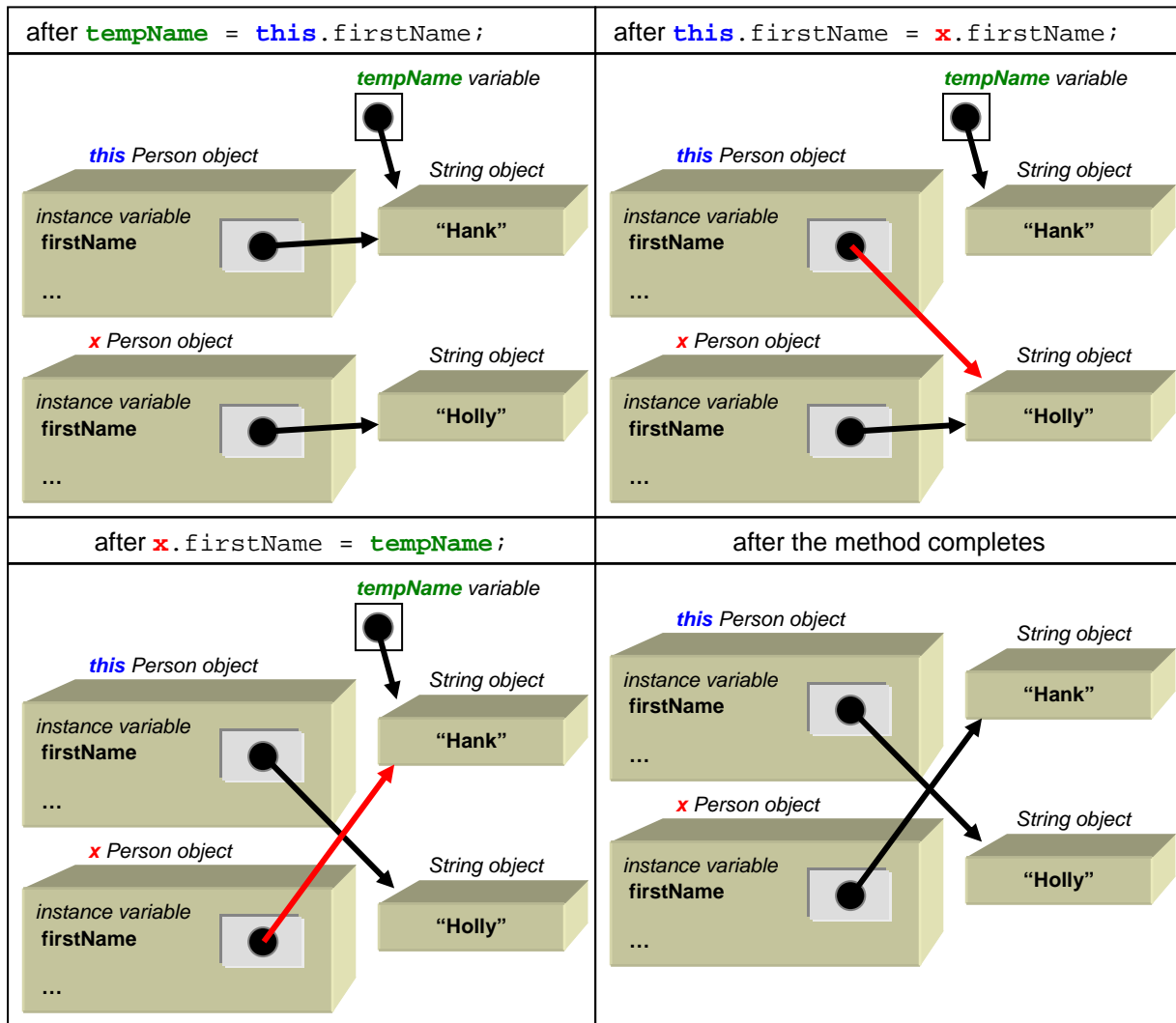
To fix this, we need a temporary variable to hold on to **this**'s **firstName/lastName** so that we don't lose it:

```
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;
}
```

Here is the diagram showing how the swapping takes place ...



You should realize that the **tempName** variable is indeed temporary. In fact, after the method completes, the **tempName** variable no longer exists. Any variables defined within a method are non-existent after the method completes ... hence the term “temporary”.

At this point lets 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).

```

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

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

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

**Our methods**

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:

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

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

        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,p3).firstName);

        System.out.println("Hank is retired ? ... " + p1.retired);
        p1.retire();
        System.out.println("Hank is retired ? ... " + p1.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
Hank is retired ? ... false
Hank is retired ? ... true
Holly's name is now: Bobby Socks
Bobby's name is now: Holly Day
```



## 4.4 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.

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

```
Person    p;  
  
p = new Person("Hank", "Urchif", 19, 'M', false);  
System.out.println(p.address.streetName);
```



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

```
class Person {  
    String    firstName;  
    String    lastName;  
    int       age;  
    char      gender;  
    boolean   retired;  
    Address   address;  
  
    Person(String fn, String ln, int a, char g, boolean r) {  
        this.firstName = fn;  
        this.lastName = ln;  
        this.age = a;  
        this.gender = g;  
        this.retired = r;  
    }  
    ...  
}
```

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 late 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).

## 4.5 Displaying an Object Using toString()

Do you recall what happens when we display one of our objects to the System console ?

```
class MyObjectTestProgram {
    public static void main(String args[]) {
        System.out.println(new Car());           // car object
        System.out.println(new Person());       // person object
    }
}
```

The result on the screen was as follows:


```
Car@19821f
Person@42e816
```

Remember that JAVA displays all of the objects that you make in this manner. What JAVA happens to be doing is converting an 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**. Hence, the following code will take a **Car** and a **Person** object, convert them to **String** objects and then display the resulting **String** objects:

```
Car        c;
Person     p;
String     s1, s2;

c = new Car();
p = new Person();
s1 = c.toString(); // s1 will be "Car@19821f"
s2 = p.toString(); // s2 will be "Person@42e816"

System.out.println("The Car as a String is " + s1);
System.out.println("The Person as a String is " + s2);
```



Notice that the output will be:

```
The Car as a String is Car@19821f
The Person as a String is Person@42e816
```

The **String** objects have the exact same characters that are displayed when we just display the objects directly using **System.out.println()**. That is because when 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 following two lines of code do exactly the same thing:

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

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.

Suppose that we wanted our **Person** object to display something like this:

Person named Hank

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

```
public String toString() {
    return ("Person named " + this.firstName);
}
```

Notice that the method has the word **public** at the front. This means that the method you are writing is publicly accessible (i.e., it can be used from anywhere in your program). This is necessary for the **toString()** method, but we will defer our discussion of **public** method access until a later time in the course.

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 be used properly by JAVA. Even the spelling and upper/lower case letters must match exactly. Then, you may notice that the method returns an actual String object that is made up of the letters "**Person named** " and then followed by the value of this **Person** object's **firstName** attribute.

What would therefore be the output of the following code:

```
Person          p1, p2, p3;

p1 = new Person();    // assume first name is set to "" within constructor
p2 = new Person("Holly", "Day", 67, 'F', true);
p3 = new Person("Hank", "Urchif", 19, 'M', false);

System.out.println(p1);
System.out.println(p2);
System.out.println(p3);
```

Here is the output ... were you correct ?

```
Person named
Person named Holly
Person named Hank
```

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

```
19 year old Person named Hank Urchif
```

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 person 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 Person named " +
            this.firstName + " " + this.lastName);
}
```

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. Lets see if we can make it into this format:

19 year old non-retired person named Hank Urchif

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

```
public String toString() {
    return (this.age + " year old " + this.retired + " person named "
        + this.firstName + " " + this.lastName);
}
```

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

19 year old false person named Hank Urchif

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. Lets 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 + " person named " +
        this.firstName + " " + this.lastName);

    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 + " person named " + this.firstName + " " +
        this.lastName;

    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 person named " +
        this.firstName + " " + this.lastName);
}
```

So, you can see that the **toString()** method may be more than one line of code but again ... the main idea is to simply keep appending to the **String** as you go ... building it up.

## 4.6 Static/Class Methods

Recall that we discussed static/class variables in a previous chapter. Class variables were used as a means of specifying attributes that were shared among all members of the class. We looked, for example, at how all **Car** objects shared the same **NUM\_WHEELS** value of **4** and how all **BankAccount** objects used the same **LAST\_ACCOUNT\_NUMBER** counter to obtain their **accountNumber** upon creation.

In JAVA, we can also create **class methods** (also known as **static methods**). Since methods are **behaviors** for the object (i.e., not *attributes*), the saving of memory space is not an issue. Therefore, the motivation for using a class/static method is different than that of using a class/static variable. Since every method must either be an *instance method* or a *class method*, we need to understand the difference.

**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/message). That is, such 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.

**Class methods**, on the other hand, are actually used (i.e., called) in a different manner, without a particular receiver object in mind. Therefore, they 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.

Recall the **computeDiscount()** method that we created for the **Person** class earlier in this chapter:

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

This is an instance method in the **Person** class which computed the discount based on the internal attributes (i.e., *gender*, *age* and *retired*) of the particular person that received the method call. So, in the code below, the receiver of the 1<sup>st</sup> call to **computeDiscount()** is **p1** and the receiver of the 2<sup>nd</sup> call to it is **p2**. It was necessary to specify the appropriate instance (i.e., Person object) before calling the method in order for the instance method to know which object to access when computing the discount.

```
Person    p1, p2;

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

System.out.println("Discount for Hank is: " + p1.computeDiscount());
System.out.println("Discount for Holly is: " + p2.computeDiscount());
```

However, we could have supplied the appropriate **Person** object as a parameter to the method as follows:

```
...
System.out.println("Discount for Hank is: " + computeDiscount(p1));
System.out.println("Discount for Holly is: " + computeDiscount(p2));
```

To make this work, we would need to re-write the method as follows ...

```
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. Therefore, it is no longer acting as an instance method, but is actually a general method that can be written and used anywhere. 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.

Although we can leave the method as it is written, it would be proper coding style to indicate to everyone that the method no longer accessed the internal attributes of the receiver object. To do this, we simply add the word **static** in the method's signature as follows:

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

Just as we specify the **class name** when we access class/static variables (e.g., **Car.NUM\_WHEELS** and **BankAccount.LAST\_ACCOUNT\_NUMBER**), we also use the **class name** to call class/static methods. So we would change our test code as follows:

```
...
System.out.println("Discount for Hank is: " + Person.computeDiscount(p1));
System.out.println("Discount for Holly is: " + Person.computeDiscount(p2));
```

In fact, the **computeDiscount()** method does not even need to be written within the **Person** class. We can, for example, make a completely different class and place such tool-like methods in there. Here, for example, is a collection of static/class methods defined in a class called **UsefulPeopleTools** that perform functions on **Person** objects that are passed in as parameters (these methods are similar to the methods that we wrote earlier in this chapter) ...



```

class UsefulPeopleTools {
    static int computeDiscount(Person p) {
        if ((p.gender == 'F') && (p.age < 13 || p.retired)) return 50;
        return 0;
    }
    static boolean isOlderThan(Person x, Person y) {
        return (x.age > y.age);
    }
    static Person oldest(Person x, Person y) {
        if (x.age > y.age) return x;
        return y;
    }
    static void swapNames(Person x, Person y) {
        String tempName;

        tempName = x.firstName;
        x.firstName = y.firstName;
        y.firstName = tempName;
        tempName = x.lastName;
        x.lastName = y.lastName;
        y.lastName = tempName;
    }
}

```

We can then use these methods by writing test code as follows:

```

class UsefulPeopleToolsTestProgram {
    public static void main(String args[]) {
        Person p1 = new Person("Hank", "Urchif", 19, 'M', false);
        Person p2 = new Person("Holly", "Day", 67, 'F', true);

        System.out.println("Discount for Hank is: " +
            UsefulPeopleTools.computeDiscount(p1));
        System.out.println("Discount for Holly is: " +
            UsefulPeopleTools.computeDiscount(p2));
        System.out.println("Hank is older than Holly?: " +
            UsefulPeopleTools.isOlderThan(p1, p2));
        System.out.println("Holly is older than Hank?: " +
            UsefulPeopleTools.isOlderThan(p2, p1));
        System.out.println("The oldest of Hank and Holly is: " +
            UsefulPeopleTools.oldest(p1, p2));

        UsefulPeopleTools.swapNames(p1, p2);
        System.out.println("Hank's name is now: " + p1.firstName +
            " " + p1.lastName);
        System.out.println("Holly's name is now: " + p2.firstName +
            " " + p2.lastName);
    }
}

```

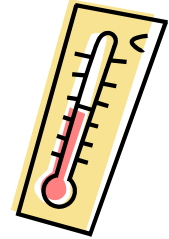
Hopefully, you will have noticed that the main difference between an instance method and a class/static method is simply in the way in which it is called. To repeat ... instance methods are called by supplying a specific instance of the object in front of the method call (i.e., a variable of the same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

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

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

Then we could use it as follows:

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