Chapter 2

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. Lastly, the idea of **static methods** is discussed.

2.1 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("5 Elm St.");
```

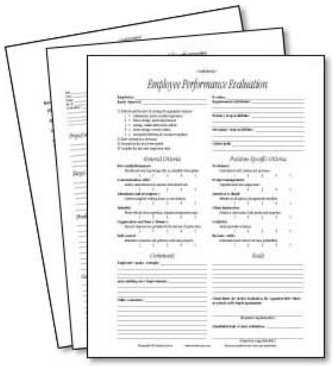
Recall that we can write a constructor for this class that allows us to provide initial values for all of the object's attributes since a constructor is a special kind of function:

```
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).

At this point in our program, we have chosen to force the user of our objects to supply parameters for ALL of the instance variables when they use (i.e., call) our constructor. So, we have taken 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 instance variables:

```
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.

2.2 Defining Methods

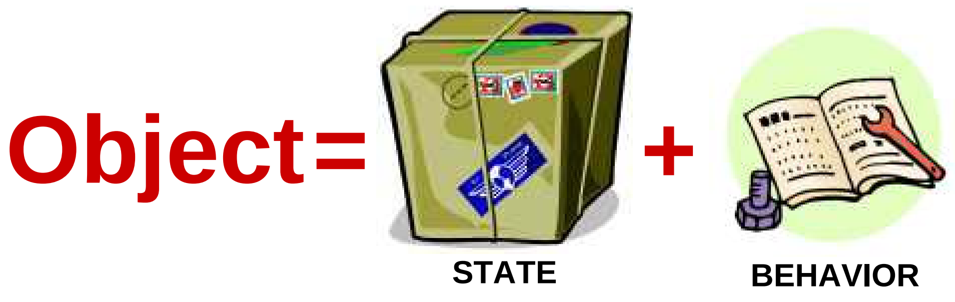
At this point you should understand that objects are used to group variables together in order to represent something called a data structure. Each object therefore has a set of *attributes* (also called *instance variables*) that represent the differences between members of the same class. For example, 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:

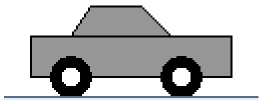
Likewise, in object-oriented programming, in addition to defining attributes, we can also define how one particular kind of object's performance and behaviors differ from another's. Defining an object's behavior is as simple as deciding what kind of functionality that the object should have. This is nothing other 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



In order to get a "feel" for what we are trying to do, it is good to compare what we did last term in Processing with what we are trying to do by defining an object's behavior. Recall, from COMP1405, the example in which we caused a car to accelerate across the screen. We defined a **Car** object and then wrote procedures to draw and move the car while having it accelerate.



Here is a comparison to how the code was organized in Processing and how it would be organized in JAVA. Note however, that the JAVA code cannot be run, as there is no **main()**:

Processing code

```
Car   myCar, yourCar;

class Car {
  int     x, y;
  float   speed;
  int     direction;
}

void setup() { ... }
void draw() {
  ...
  drawCar(myCar);
  drawCar(yourCar);

  moveCar(myCar);
  moveCar(yourCar);
}

void drawCar(Car aCar) {
  ...
}

void moveCar(Car aCar) {
  ...
}
```

JAVA code

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

  public void draw() { ... }
  public void move() { ... }
}
```

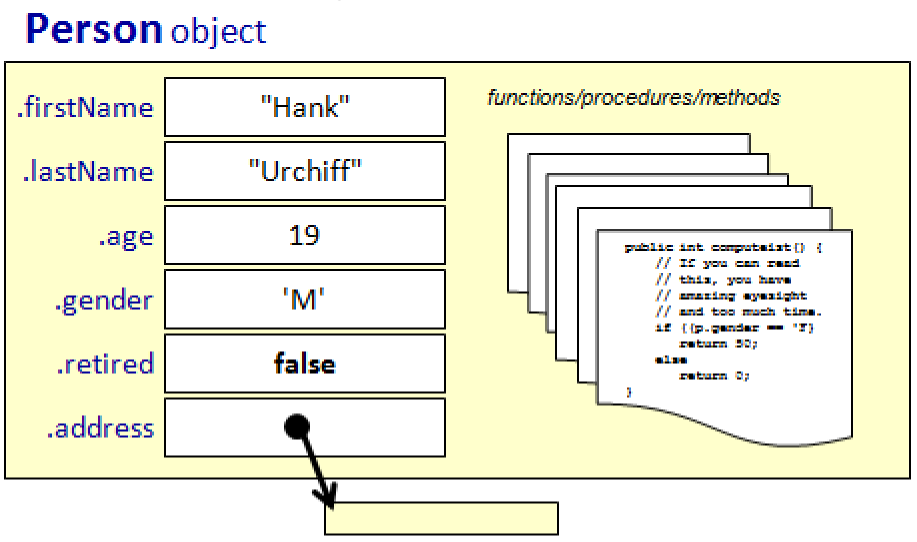
```
// This code is not runnable, no main()
public class AccelProgram {
  Car      myCar, yourCar;

  void setup() { ... }
  void draw() {
    ...
    myCar.draw();
    yourCar.draw();

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

In the Processing code, you may notice that the **drawCar()** and **moveCar()** procedures take in a **Car** object as a parameter. That is, these procedures require a **Car** object in order to work. In fact, since these procedures work on whatever car is passed in as a parameter, each procedure itself represents a unique behavior for **Car** objects ... the ability to move and the ability to draw itself.

To convert this code to object-oriented style JAVA code, we define the car in its own class and then move the **drawCar()** and **moveCar()** procedures into that class as well. As a result, the **Car** class will contain code to **move()** and **draw()** ... that is ... each **Car** object knows how to move and draw itself. Notice how the program itself then becomes much simpler.

Notice how the parameter is not used anymore but instead we "call" the procedure by using the car object itself, followed by the dot operator. This is similar to what we did to access an object's attributes ... because the procedures are now "inside" the **Car** object's class definition just like the attributes are defined inside there as well.

What does all of this mean ? We will write functions and procedures (commonly referred to as ***methods*** in JAVA) **within our class** definitions along with the attributes. So we can think of an object as being a set of attributes as well as a set of methods all included "inside" the class →



```
public int computeIt() {
    // If you can read
    // this, you have
    // amazing eyesight
    // and too much time.
    if ((p.gender == 'F')
        return 50;
    else
        return 0;
}
```

Let us create a method in JAVA. Consider the **Person** class again. 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 the instance variables (i.e., 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;
        else
            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 (an 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');
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());
```

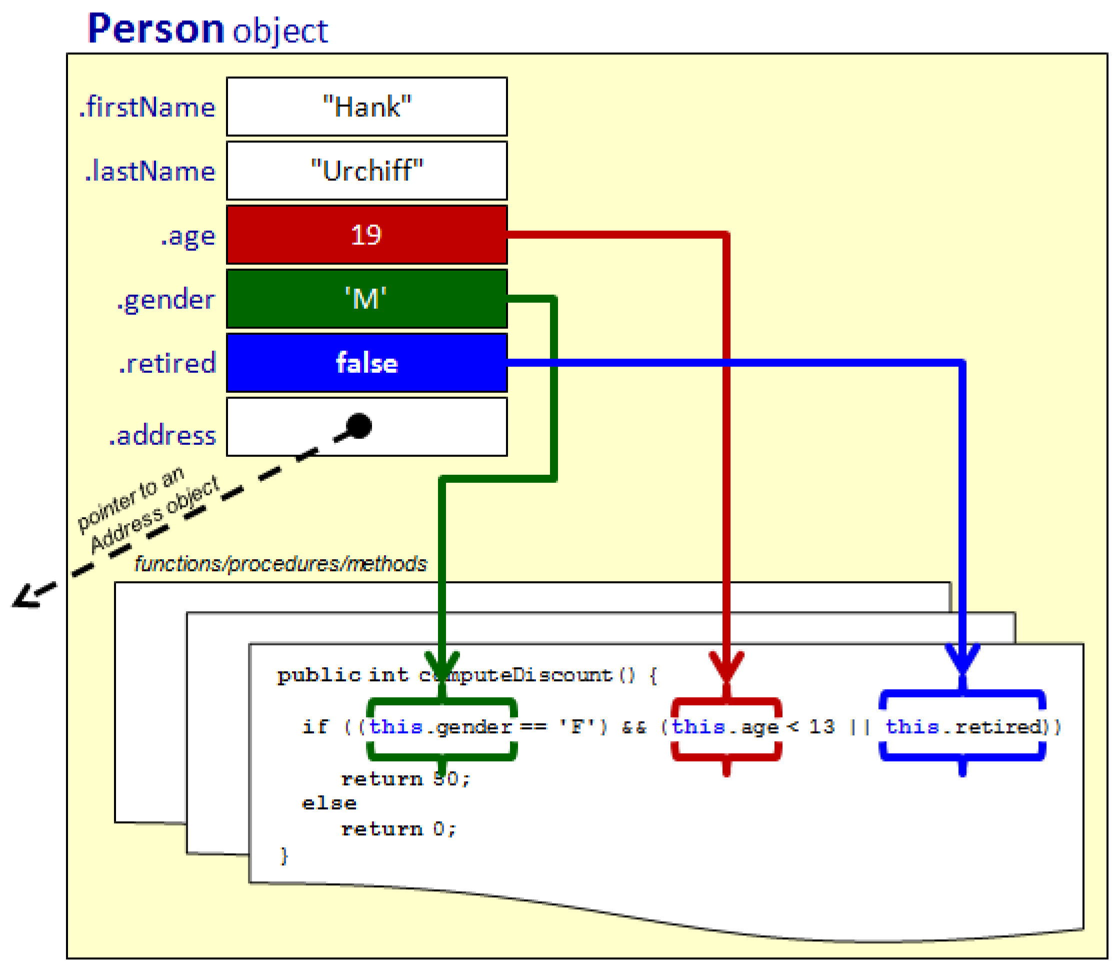
As it turns out, if you leave off the keyword **this**, JAVA will "assume" that you meant the object that received the method call in the first place and will act accordingly. Therefore, the following code is equivalent and often preferred since it is shorter:

```
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:



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

Let us write 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;
}
```

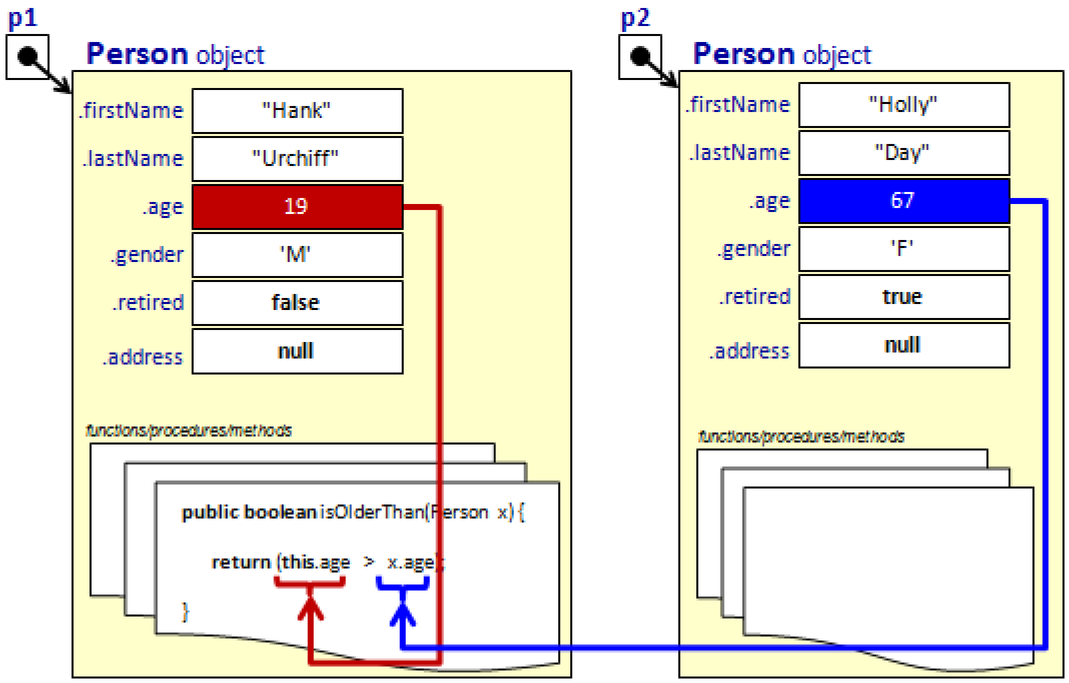
... or the more efficient version:

```
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", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F');
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)**:



As you can see, 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');
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 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).

```
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
    int computeDiscount() { ... }
    boolean isOlderThan(Person  x) { ... }
    Person oldest(Person  x) { ... }
    void retire() { ... }
    void swapNameWith(Person x) { ... }
}
```

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 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');
        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
```

2.3 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.

Lets 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) {
        this.firstName = fn;
        this.lastName = ln;
        this.gender = g;
        this.retired = r;
        this.age = 0;
        this.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 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).

2.4 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 **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 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) { ... }
```

2.5 *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');

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 an 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 *instance* method	Using it as a *class* method
`Person     p1, p2, p3;`	`Person     p1, p2, p3;`
`Toolkit    toolkit;`	`p1 = ...;`
`toolkit = new Toolkit();`	`p2 = ...;`
`p1 = ...;`	`p3 = ...;`
`p2 = ...;`	`S.o.p(Toolkit.computeDiscount(p1));`
`p3 = ...;`	`S.o.p(Toolkit.computeDiscount(p2));`
`S.o.p(toolkit.computeDiscount(p1));`	`S.o.p(Toolkit.computeDiscount(p3));`
`S.o.p(toolkit.computeDiscount(p2));`	
`S.o.p(toolkit.computeDiscount(p3));`	

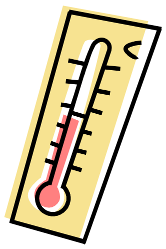
This is the essence of a class/static method … the idea that the method does not necessarily need to be called by using an instance of the class.

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.