CS125 : Introduction to Computer Science

Lecture Notes #18
`static` versus non-`static`, and Constructors

Lecture 18 : `static` versus non-`static`, and Constructors

Some more terminology

- *instance* – an object. Every object is an instance of a particular class.
- *instantiation* – the allocation of an object. We have a class, and we "allocate an object of that class", or "instantiate an instance of that class".
- The two kinds of methods:
  - *instance method* – does NOT have `static` on its first line; this method is tied to the instance – i.e. you choose an instance, run the method on an instance and it operates on the variables of that instance and no other, until that method call has completed. `printClock()` from the last lecture is an instance method of the class `Clock`; when you called the method `printClock(...)`, you indicated what reference's object it should manipulate, by listing that reference before the dot in front of the method call, and then the method call manipulated that object (i.e. its `this` reference pointed to that object). If you do not specify which instance the method should manipulate, then you cannot call the method (i.e. the method call is a syntax error unless you have a reference and dot in front of it)
  - *class method* – has `static` on its first line; this method is not tied to any particular instance, but rather is just a helpful method that we've put in this class because it's related to this class. `Keyboard.readInt()` is a class method.
- The four kinds of variables:
  - *local variable* – a variable declared using a variable declaration statement within the curly braces of a method
  - *parameter variable* – a variable "declared" in the parameter list of a method (the `this` variable of an instance method gets included in this category, though the compiler declares it automatically, without you needing to write code to do so)
  - *instance variable* – a variable declared using a variable declaration statement within a class but not within any method, and without the keyword `static` on the declaration line; such a variable is tied to an instance – i.e. each instance of that class has its own version of the instance variable, and two different instances will each have their own, independent version of an instance variable, rather than sharing the same variable. In the last three notes packets, `hour`, `minutes`, and `AM` are instance variables of the class `Clock`. (In a sense, the indexed array cells of an array are also instance variables, though since they are "named" with indices instead of names, we don't usually call them "instance variables".)
  - *class variable* – a variable declared using a variable declaration statement within a class but not within any method, and with the keyword `static` on the declaration line; there is one copy of this variable, that can be accessed from anywhere in the program, through the class name itself – rather than one copy per instance, that gets accessed through a reference to that instance. (We'll talk more about class variables in a moment.)

So, we saw last time that instance methods and class methods basically did the same thing, just via different syntax. So, why might you use one over the other? Well, basically, instance methods place the "focus' on your data instead of on the method. That is to say, there's no real difference in the machine between the two. They are just two different sets of syntax for accomplishing the same thing – it's just that the syntax for instance methods lets you think of a method as something that runs on an object, rather than thinking of an object as something you send to a method. So the use of instance methods can change the way you view your program – you tend to think of your data first, rather than thinking about your subroutines first. That's a subtle difference, so for now, we'll just say that sometimes class methods are more convenient, and other times instance methods are more convenient. You won't have to choose which is the better way in this course, but you do want to understand the syntax for writing and using both kinds of methods. Generally, though, we will make something an instance method if we can.

Another way to look at the instance method syntax is that it makes calling instance methods consistent with using instance variables. If you have an *instance variable* or *instance method*, you access it by using a reference to that instance (i.e. a reference to that object), followed by a dot, followed by the variable or method. For example:

```
Clock c1 = new Clock();
c1.hour = 9;
c1.minutes = 53;
c1.AM = true;
c1.printClock();
```

If you have a *class variable* or *class method*, you access it by using the class name, followed by a dot, followed by the variable or method. For example, in the `Math` class you might see the following:

```
public static double PI = 3.1415926;
public static double cos(double x) {...}
```

and you'd use those things as follows:

```
double myVal = Math.PI;
double cosOfMyVal = Math.cos(myVal);
```

You *can* have classes with a mix of class variables and methods, and instance variables and methods, but we won't do that too often in this course – our classes will tend to have only "class stuff" (i.e. "`static` stuff") or only "instance stuff" (i.e. "non-`static` stuff").

Constructors

Constructors are instance methods used to initialize an object when it is first created.

- Same name as class
- No return type
- When object is created, some constructor is always called. (what if you don't write one? More on that later.)

Let's look at the `Clock` class again...but this time let's leave out the instance method `SetTime` (we could have kept it as well, but this way the class will still fit on one slide) and add a constructor which does the same thing as `SetTime` did).

```
public class Clock
{
   public int hour;
   public int minutes;
   public boolean AM;

   public Clock(int theHour, int theMinutes,
                boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   } // but you don't *need* the this. part

   public void printClock()
   {
      System.out.print("Time is " + this.hour + ":");
      if (this.minutes < 10)
         System.out.print("0");
      System.out.print(this.minutes + " ");
      if (this.AM == true)
         System.out.println("AM.");
      else    // AM == false
         System.out.println("PM.");
   }
}  // end of class
```

```
public class ClockTest
{
   public static void main(String[] args)
   {
      // declare reference variables
      Clock home;
      Clock office;

      // create "home" object;
      //    initialize "home" reference
      home = new Clock(2, 15, true);

      // create "office" object;
      //    initialize "office" reference
      office = new Clock(7, 14, false);

      // call instance method to print
      //   instance variables
      home.printClock();
      office.printClock();
   }
}
```

The expression:

```
new Clock(2, 15, true)
```

or, in fact, any object allocation (except for an array), is a three-step process:

```
new Clock(2, 15, true)        object is
---------                       allocated
   (1)

new Clock(2, 15, true)
    ------------------      object is
            (2)             initialized
                            using constructor

new Clock(2, 15, true)
----------------------     address of
            |               object in memory
address <------|   (3)       is returned
```

and then of course that returned address would be stored in a reference if the expression `new Clock(2, 15, true)` were used in a line like this:

```
Clock c1 = new Clock(2, 15, true);
```

Method Overloading

You can have multiple constructors in one class. Having two methods with the same name (but different parameter lists) is known as *method overloading*, since we are overloading the method name with multiple definitions. If the name is the same, the compiler can only figure out which method we want by comparing parameter lists, so the parameter lists must then be different. This can be done for constructors or for any other methods. When you call the `System.out.println(...)` methods, the same idea exists there – there are many different versions of that method, all with different parameter lists. The method with no parameters, matches the case where you have no arguments in your `System.out.println()` call. The method with one integer parameter, matches the case where you have one integer argument for your method call. And so on.

```
public class Clock
{
   public int hour;
   public int minutes;
   public boolean AM;

   // a constructor
   public Clock()
   {
      this.hour = 12;
      this.minutes = 0;
      this.AM = true;
   }

   // another constructor
   public Clock(int theHour, int theMinutes,
                boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }

   // a third constructor
   public Clock(Clock c)
   {
      this.hour = c.hour;
      this.minutes = c.minutes;
      this.AM = c.AM;
   }
```

```
   // a non-constructor instance method
   public void printClock()
   {
      System.out.print("Time is " + this.hour + ":");
      if (this.minutes < 10)
         System.out.print("0");
      System.out.print(this.minutes + " ");
      if (this.AM == true)
         System.out.println("AM.");
      else    // AM == false
         System.out.println("PM.");
   }

   // another non-constructor instance method
   public void setTime(int theHour, int theMinutes, boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }
}  // end of class

public class ClockTest
{
   public static void main(String[] args)
   {
      // declare reference variables
      Clock home;
      Clock office;
      Clock car;

      // create objects; initialize references
      home = new Clock(2, 15, true);
      office = new Clock();
      car = new Clock(home);

      // change time on "home" clock
      home.setTime(8, 13, false);

      // call instance method to print
      //   instance variables
      home.printClock(); // Time is 8:13 PM.
      office.printClock(); // Time is 12:00 AM.
      car.printClock(); // Time is 2:15 AM.
   }
}
```

Recall that before we introduced constructors, we still had lines like

```
home = new Clock();
```

This is because the system provides a *default constructor* if we don't write one. This provided constructor would be empty:

```
public Clock()
{
   // nothing of value here
}
```

but at least there is *some* constructor to call, then. That is why the code in lecture packets 15 through 17 would compile – the compiler would have at least given us an empty, useless no-argument constructor so that some constructor exists to call, since there has to be *some* constructor called everytime a (non-array) object is allocated.

If there are any constructors given, no default is provided. So, if you want two constructors and one of them is the no-argument constructor, you have to write the no-argument constructor. You can't write a 2-argument constructor and assume the no-argument one will be added by default. You only get the no-argument constructor by default if you have *no constructors at all* in your source code.