CS125 : Introduction to Computer Science

Lecture Notes #17
Instance Methods

Lecture 17 : Instance Methods

During the last lecture, we used the following code to make use of a class:

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

      // allocate objects and assign values to reference variables
      home = new Clock();
      office = new Clock();

      // set the times on the clocks
      setTime(home, 2, 15, true);
      setTime(office, 7, 14, false);

      // print the clocks
      printClock(home);
      printClock(office);

   }  // end main


   public static void setTime(Clock c, int theHour, int theMinutes,
                                                boolean theAM)
   {
      c.hour = theHour;
      c.minutes = theMinutes;
      c.AM = theAM;
   }

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

And this was the class we were using:

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

We're going to start changing that code a bit. First of all, why do we have `setTime(...)` and `printClock(...)` in the `ClockTest` class? If we wanted to use the `Clock.java` file in the future, we would copy `Clock.java` into the directory where the rest of our new project was. But, if we did that, we still wouldn't have `setTime(...)` and `printClock(...)` available to us, since those methods are not in `Clock.java`. We'd have to either also copy `ClockTest.java` into our directory as well...or else we'd have to open `ClockTest.java` and copy `setTime(...)` and `printClock(...)` so that they could be pasted into some other file in our project directory. Either way, working on our new project requires us to deal with both `Clock.java` and `ClockTest.java`, since the code we want is scattered across both those files.

Wouldn't it be easier if those two methods were in the `Clock.java` file? After all, the purpose of those two methods is to manipulate `Clock` objects, so we'd never use those methods without also needing the `Clock.java` file to provide the instance variables. If we had the instance variables *and* those two methods, in the `Clock.java` file, then when we copy the `Clock.java` file to a new directory, we are copying not just the blueprint for `Clock` objects, but also, we are copying the methods for manipulating `Clock` objects. Keeping all the `Clock`-related code in one file, makes more sense than does scattering that code across two or three or four files. This way, to use `Clock` objects, you would only have to copy one file – `Clock.java` – which contains *all* the code that pertains to `Clock` objects – the instance variable declarations that form the blueprint for those objects, and the methods that use those objects.

That would lead to the following `Clock.java` file:

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

   public static void setTime(Clock c, int theHour, int theMinutes,
                                              boolean theAM)
   {
      c.hour = theHour;
      c.minutes = theMinutes;
      c.AM = theAM;
   }

   public static void printClock(Clock c)
   {
      // print variables for clock
      System.out.print("Time is " + c.hour + ":");
      if (c.minutes < 10)
         System.out.print("0");
      System.out.print(c.minutes + " ");
      if (c.AM == true)
         System.out.println("AM.");
      else    // AM == false
         System.out.println("PM.");
   }

}
```

The `Clock` class above is the same as the `Clock` class from the earlier example, except that now we have also moved the two non-`main()` methods from `ClockTest.java`, into `Clock.java`, as well.

That would mean that in the `ClockTest.java` file, the only method you'd have left is `main(...)`. And the code within `main(...)` can remain exactly the same, with one exception – since now, when the methods `setTime` and `printClock` are called, those methods are in a different class than `ClockTest`, you need to put that different classname in front of the method call, followed by dot. That is, rather than a method call such as:

```
 printClock(home);
```

you now need to have:

```
Clock.printClock(home);
```

This is no different than how, in your own MPs, you need to use the expression `Keyboard.readInt()`, rather than just the expression `readInt()`, to input an integer.

Thus, the `ClockTest.java` file now looks like this:

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

      // allocate objects and assign reference variables
      home = new Clock();
      office = new Clock();

      // set the times on the clocks
      Clock.setTime(home, 2, 15, true);
      Clock.setTime(office, 7, 14, false);

      // print the clocks
      Clock.printClock(home);
      Clock.printClock(office);

   }  // end main
}  // end of class
```

Note that the `main()` method simply uses existing methods to manipulate the `Clock` objects, and has no need to directly read or write the `hour`, `minutes`, or `AM` variables of the `Clock` objects.

Now, consider for a moment the `Clock` class, as it is currently written. We have two methods, `setTime(...)` and `printClock(...)`, both of which have a `Clock` reference as a parameter. What if we had many more methods in `Clock.java`, all of which manipulated `Clock` objects? For example, in addition to `setTime(...)` and `printClock(...)`, we might have a method `incrementOneMinute(...)` which would move a `Clock` object's time forward one minute. We might have a method `changeToDST(...)` that would convert the time from "standard time" to "daylight savings time". You can imagine many other methods you might write, all of which would be designed to write or read the instance variables of a `Clock` object. And thus, each of these methods would need a `Clock` reference as a parameter, since there's no way a method could manipulate a `Clock` object, unless it had access to a `Clock` reference that pointed to that object.

So, imagine you had 100 different methods in the `Clock.java` file, all of them designed to manipulate `Clock` objects in some way, just like `setTime(...)` and `printClock(...)` do – and thus all of them having a `Clock` reference as a parameter, just like `setTime(...)` and `printClock(...)` do. In such a case, you could argue that the need for a `Clock` reference as a parameter in each and every one of those 100 methods, is a little bit annoying. After all, of *course* each of these 100 methods needs access to a `Clock` reference. That's why they are in the `Clock.java` file in the first place – because they manipulate `Clock` objects! It seems like it's a bit redundant to then have to say, for every single method in that file, "Oh, and this method needs a `Clock` reference too!". It would be nice if every method in the `Clock.java` file automatically had a `Clock` reference, without us having to list it explicitly – since we wouldn't put a method in the `Clock.java` file in the first place unless we also felt it needed a `Clock` reference as a parameter.

Can we do this? Is there some facility in the Java language for saying "Assume every method in this class has a `Clock` parameter, so that I don't have to list that parameter in each method!"?

The answer is, yes, there is! (Well, almost. That's not *quite* what we're allowed to do, but what we're allowed to do will be good enough.)

The syntax for accomplishing this is what we will look at next. However, before we begin, a word of warning – the earlier code examples in this packet compiled, and the final pair of `ClockTest.java` and `Clock.java` files in this packet will compile, but the intermediate steps will not. That is, we'll be making four changes to the last code example, to produce our new code example, and *all four* changes have to be made. So, when we've made only one of the changes, or two, or three of them, we will show you the "code so far", but those examples won't compile. It will only be once we've made all four changes, that the resultant code will compile. That will be the last `Clock.java` and `ClockTest.java` files in this packet.

The first of our four changes, will be to change the parameter name in the two methods in `Clock.java` from `c` to `this` (and thus we will change all the code that uses that parameter name, as well):

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

   public static void setTime(Clock this, int theHour, int theMinutes,
                                              boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }

   public static void printClock(Clock this)
   {
      // print variables for clock
      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.");
   }
}
```

The new parameter name specifically has to be `this`; the variable name `this` is a reserved word in Java, that is specifically used for the sorts of situations we are talking about now, and which cannot be used as the name of any other kind of variable – `this` can *only* be the name of a parameter we are trying to have the compiler assume "automatically exists", as we are in the process of discussing right now.

The second change we want to make, is to remove the word `static` from the first line of each of the two methods. The word `static` is basically a signal to the compiler – when the word is there, one signal is sent to the compiler, and when the word is not there, a different signal is sent to the compiler. So, `static` doesn't have any inherent meaning itself; it's the *presence or lack* of the word `static` that is important.

Up to now, every method we've written has had `static` on it's first line; such methods are called *class methods.* Now, for the first time, we are writing methods that do NOT have `static` on the first line; such methods are called *instance methods.* Here is the code so far, once we have removed `static` from the first line of each of the two methods:

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

   public void setTime(Clock this, int theHour, int theMinutes,
                                              boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }

   public void printClock(Clock this)
   {
      // print variables for clock
      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.");
   }
}
```

So, what is the difference between an instance method and a class method? Well, fundamentally, they are the exact same thing – in both cases, we are simply making use of procedural abstraction, i.e. calling a method, passing it some values, and perhaps getting a value back when the method has completed. The differences between the two are syntax-related; instance methods make use of slightly different syntax to accomplish the same things that class methods accomplish. However, the use of that slightly different syntax will result in us thinking about instance methods in a slightly different way than we think about class methods, and that slight change in our thought process is what we are most concerned about. We'll elaborate on that more in the next packet; for now, just think of an "instance method" as a slightly different syntax for doing the same thing as a "class method".

One of the advantages an instance method has over a class method, is that we no longer specifically need to list the `Clock` parameter in our method signatures. In fact, not only are we allowed to not list it, but in fact, we are *required* to not list it; removing the `static` from the first line of the method, means that the compiler will declare a `Clock this` parameter for us, and so we should not specifically do so ourselves. Removing that parameter from the parameter list of both methods, is our third change, and it leads to the following code:

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

   // There is a "Clock this" parameter here automatically, because
   // setTime(...) is an instance method
   public void setTime(int theHour, int theMinutes, boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }

   // There is a "Clock this" parameter here automatically, because
   // printClock(...) is an instance method
   public void printClock()
   {
      // print variables for clock
      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.");
   }
}
```

Now, with the above change, our Clock.java file is correct. Both methods in that file are now correctly-written instance methods. The fourth change we need to make, happens in our ClockTest.java file, where we actually *call* the instance methods. Not only is the instance method syntax slightly different from the class method syntax, but also, the syntax for calling an instance method, is slightly different from the syntax for calling a class method.

The difference in syntax, is related to our now-removed parameter in the methods in Clock.java. Without that parameter, we should not be sending in an argument, since the argument list always needs to match with the parameter list. That suggests we should change our method call to setTime(...) (for example) from the following (which is what we have now):

```
Clock.setTime(home, 2, 15, true);
```

to the following:

```
Clock.setTime(2, 15, true);
```

Now, we have three arguments – two int values and one boolean value, in that order – and those three arguments match our parameter list from the new version of setTime(...) in our Clock.java file:

```
// There is a "Clock this" parameter here automatically, because
// setTime(...) is an instance method
public void setTime(int theHour, int theMinutes, boolean theAM)
{
   this.hour = theHour;
   this.minutes = theMinutes;
   this.AM = theAM;
}
```

However, we still have a problem. Our this parameter in setTime(...) might be automatically given, but we still need an "argument" for it. Specifically, we need to know if this should point to the same object as home points to, or if instead it should point to the same object as office points to. With class methods, we made this clear by passing home or office as an argument in the argument list, but since with instance methods, the this parameter is not in the regular parameter list, we also don't want to list its argument in the regular argument list. So where do we put the argument home or office?

The answer is, we move that argument to the front of the method call, and place a dot after it. That is, we move to this syntax:

```
home.Clock.setTime(2, 15, true);
```

The argument home – whose value we want to copy into the automatic parameter this – gets placed in the front of the method call. And the other arguments – whose values get copied into the parameters we actually listed in the method's parameter list – get placed in the actual argument list within the parenthesis, as usual.

There is one more change left to make. We do not need both home and Clock in front of the method call. We've already explained that home *has* to go there. But home not only points to the object that we want this to point to, but in addition, since home is of type Clock, that tells the compiler to look in the Clock class for this method, and so we do not need to explicitly list the typename "Clock" after home in the method call. In fact, we don't even have the option of leaving it there if we want – since the extra use of the typename "Clock" is redundant, we *have to* remove it. This leaves us with the final version of this method call:

```
   home.setTime(2, 15, true);
```

And then the other three method calls in `main()` are likewise changed in this manner:

```
   Clock.setTime(office, 7, 14, false);  ----> office.setTime(7, 14, false);
   Clock.printClock(home);               ----> home.printClock();
   Clock.printClock(office);             ----> office.printClock();
```

And that change – which is our fourth and final change – gives us the following version of `ClockTest.java`:

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

      // allocate objects and assign reference variables
      home = new Clock();
      office = new Clock();

      // set the times on the clocks
      home.setTime(2, 15, true);
      office.setTime(7, 14, false);

      // print the clocks
      home.printClock();
      office.printClock();

   }  // end main
}  // end of class
```

which compiles together with the final version of `Clock.java` we had earlier:

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

   // There is a "Clock this" parameter here automatically, because
   // setTime(...) is an instance method
   public void setTime(int theHour, int theMinutes, boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }

   // There is a "Clock this" parameter here automatically, because
   // printClock(...) is an instance method
   public void printClock()
   {
      // print variables for clock
      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.");
   }
}
```

Now that those four changes have been made, we have converted our example over from using class methods, to using instance methods. The two examples work basically the same way; the instance method version just has slightly different syntax.

For example, consider this code snippet from our newest version of `ClockTest.java`:

```
             home.printClock();
```

What we are doing here is invoking `printClock` off a `Clock` reference. Before we actually enter the `printClock` method, our memory looks something like this:

```
-----------------------------
|                           |
|  main                     |          ________
|                           |          |      |
|              home --------|--------> |Clock |
|                           |          |object|
|                           |          |______|
|                           |
|                           |
|___________________________|
```

Once the method call actually begins, our memory looks something like this:

```
-----------------------------
|                           |
|  main                     |          _______
|                           |          |      |
|            home ----------|--------> |Clock |
|                           |          |object|
|                           |          |______|
|                           |            ^
|     ______________________|_______     |
|     |                            |     |
|     |   printClock     this -----|-----|
|_____|                            |
      |                            |
      |                            |
      |____________________________|
```

That is, `this` points to the same object that `home` points to. The reference variable `home` holds the machine address where the `Clock` object is located (which is why we say `home` "refers" to the object), and that machine address is copied into `this`, so that `this` now also holds the machine address where the `Clock` object is located (which is why we say that `this` now "refers" to the `Clock` object too). The class method version of this program worked the same way, except that (1) we called the parameter in `printClock(...)` "`c`" instead of "`this`", and (2) we actually sent `home` as an argument in that case, to an actual parameter `c`, instead of having our argument in front of the method call and our parameter automatically created by the compiler.

Of course, in these examples, `this` is of type `Clock` only because the instance method is in the `Clock` class. If the instance methods were in the `String` class, the `this` in each instance method would be of type `String`, for example. The type of an instance method's `this` reference matches the type the instance method is a part of (i.e. matches the class the instance method is in).

Taking advange of the this default use

One nice thing about the use of this is that it is often assumed by default. For example, our printClock() instance method could also have been written as follows:

```
public void printClock()
{
   System.out.print("Time is " + hour + ":");
   if (minutes <= 9)
      System.out.print("0");
   System.out.print(minutes + " ");
   if (AM == true)
      System.out.println("AM.");
   else    // AM == false
      System.out.println("PM.");
}
```

We *can* put the this. in front of hour, minutes, and AM if we want, but we don't have to – if we leave the this. off, the compiler assumes that we meant to put it there anyway, and so it will put it in for us. For that reason, instance methods are usually written as you see directly above, without the explicit use of this. in front of the instance variables of the class. After all, if the compiler will put it in for you anyway, why bother typing it in yourself? That is another way that using instance methods makes our life as programmers a little bit easier. (Now, in CS125, we will always write this. where it is needed, just to remind you what is going on. However, you are not required to do so yourself.)

The one exception to this would be if you have a parameter or local variable with the same name as your instance variables:

```
public void printClockStrangeExample(int hour)
{
   int minutes = 10;
   System.out.print("Time is " + hour + ":");
   if (this.minutes <= 9)
      System.out.print("0");
   System.out.print(minutes + " ");
   if (AM == true)
      System.out.println("AM.");
   else    // this.AM == false
      System.out.println("PM.");
}
```

In the code above, when hour is printed to the screen, it will be the parameter hour, not the instance variable hour. Likewise, when minutes is printed to the screen, it will be the local variable minutes which we have assigned the value 10. When the compiler sees a variable in a method, the order it checks things in is as follows:

1. First, it sees if this was a local variable declared in this method.
2. If not, it sees if it was a parameter variable for the method.
3. If it is not either of the above two, then, if the method is a class method, there are no other options and the compiler will alert you to an error. If the method is an instance method, however, there is one additional possibility – that the variable is an instance variable of the class the method is in, and that we are trying to access that instance variable but just didn't bother to put a `this.` in front of the variable name.

The version of `printClockStrangeExample` below shows the above code, changed so that the `hour` and `minutes` that are printed to the screen are the instance variables and not the parameter and local variable. Note that we still can avoid putting a `this.` in front of `AM`, since there's no confusion there. But we need the `this.` in front of `hour` and `minutes` now to make it clear that we want the instance variable version, rather than assuming the use of `minutes` is the local variable (choice 1 in the list above) or assuming that the use of `hour` is the parameter variable (choice 2 in the list above).

```
public void printClockStrangeExample(int hour)
{
   int minutes = 10;
   System.out.print("Time is " +
                this.hour + ":");
   if (this.minutes <= 9)
      System.out.print("0");
   System.out.print(this.minutes + " ");
   if (AM == true)
      System.out.println("AM.");
   else    // this.AM == false
      System.out.println("PM.");
}
```

Note that having local variables with the same names as your parameters, or having local variables or parameter variables with the same names as your instance variables, is NOT a good idea, due to the fact that it can cause exactly the confusion we have described above. So this issue tends to not really come up since usually you give your parameter variables and your local variables different names than you give to your instance variables. And since this issue doesn't come up often, as a general rule you can just leave the `this.` off and the compiler will assume you meant to put it in, and everything will be fine.

Instance variables in memory

The machine is able to access the instance variables of an object precisely *because* it knows the starting address of the object. When you have a line of code such as:

```
home.printClock();
```

as we have discussed, the starting address of the object is what is located in the reference variable. So, we know where the overall object begins.

So the compiler, when compiling the code for the method, defines all the accessing of an object in terms of the starting address of the object. If some instance variable within the object (such as `minutes`, for objects of type `Clock`) is always four cells from the beginning of the object, when once you know the starting address of the object, you simply move four more cells downward and there is the particular instance variable you are looking for. If the object begins at `a60`, then your instance variable (such as `minutes`) is at `a64`. If your object instead begins at `a10084`, then your instance variable that is four cells from the start, instead begins at `a10088`. So we need the reference variables – they tell us the starting address of the object, and that is the only information we are missing!