CS125 : Introduction to Computer Science

Lecture Notes #19
Access Permissions and Encapsulation

Lecture 19 : Access Permissions and Encapsulation

Access Permissions

Since we don't actually need to access the three variables, let's hide them away!

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

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

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

   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.");
   }
}
```

- `public` means accessible to *client* code – the code which declares references to this type and allocates objects of this type.
- `private` means accessible only to this class and its instance methods and instance variables
- (If instead of having `public` or `private`, you have no access permission listed at all, then that implies *package* access. We will not be using that in this course – we will always have some specific access permission keyword instead of leaving one out entirely.)
- The collection of method signatures for the `public` instance methods (along with documentation telling the client what those methods do) is called the *interface* of a class. The code which makes the interface work – the instance variables, the definitions for the `public` instance method signatures, and any `private` instance methods you have written – is called the *implementation*.
- By keeping the instance variables `private`, the implementation is hidden from the client! This is called *data hiding* or *encapsulation*. We force the client to use only the `public` methods – the interface – to access an object.
- The question is, why would we want to do this? More on that in a minute. First, an example.

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

      // allocate and initialize new clocks,
      //   point references to those clocks
      home = new Clock(2, 15, true);
      office = new Clock();

      // print variables for clock at home
      home.printClock();

      // print variables for clock at office
      office.printClock();

      // these two lines will cause compiler
      //   errors because variables are private
      home.AM = true;
      office.minutes = 34;
   }
}
```

What you are restricting is access to the *name* – the name of a variable or name of a method. When we say that the line:

```
    home.AM = true;
```

will cause compiler errors, we are not saying that `home.AM` does not exist. Quite the contrary – `home` does indeed point us to a `Clock` object, and that `Clock` object does indeed have two integer variables and a boolean variable, just like any other `Clock` object. However, those variables are somewhere in memory. We have no idea where the object is, and we thus have no idea where its variables are. The only way for *us* to get to those variables is by using Java syntax such as the line above. The code we run knows where the object is, because of `home`, and it knows where `AM` is within that object, because it knows the definition of `Clock`.

So there is no problem with using the line above, in theory. When we make our `AM` instance variable `private`, we are saying, "have the compiler complain about this, for no other reason than that we want the compiler to complain about it". Nothing changes in memory as a result of using `private` instead of `public`. All that changes is that now, instead of the compiler letting us do whatever we want to the `Clock` object variables, the compiler now complains instead when we use the `Clock` variables.

Imagine if we had said:

```
   int x;
```

and then every time we wanted to use `x`, the compiler gave us an error message and said, "Sure, `x` does exist, it is a variable, and it stores a value, but I don't want you using `x`. Go away." That's the idea of making something `private`.

Why do this? Why hide the implementation from the client so that the client cannot use it directly?

1. Too much flexibility can be dangerous – why offer it if you don't want the client to have it?

2. If we use instance variables directly, and change the collection of instance variables we use to implement the class, then we have to find and change all code that used the old collection of instance variables, and make it use the new collection of instance variables instead. Why put ourselves in the position of having to do so much work?

3. A new programmer doing code maintenance has to understand a lot of code written using various specific instance variables. Why put the maintenance programmer in that position?

How does hiding implementation solve these problems?

1. Too much flexibility can be dangerous – why offer it if you don't want the client to have it?

   If the client can only invoke the instance methods you provide with the class, but cannot access the instance variables directly, then the client can only read and alter the instance variables in the ways that you decide "make sense" ahead of time.

2. If we use instance variables directly, and change the collection of instance variables we use to implement the class, then we have to find and change all code that used the old collection of instance variables, and make it use the new collection of instance variables instead. Why put ourselves in the position of having to do so much work?

   Client code uses only the interface. The interface must be well-designed and thought out before we offer the class to the world. But once we do, then we can change the implementation as much as we want but client code doesn't break unless we change the interface (because client code doesn't use the implementation directly). (See the end of the packet for a re-implementation of the `Clock` class.)

3. A new programmer doing code maintenance has to understand a lot of code written using various specific instance variables. Why put the maintenance programmer in that position?

   The maintenance programmer can read code which uses only the method calls which are part of the interface. These are given intuitive names to make code easier to understand. Thus client code focuses on setting big ideas into motion, and the details are hidden in the implementation and the maintenance programmer doesn't necessarily need to know them.

- In addition, throw in one more useful idea of classes in general. If all your data is declared in `main`, it is hard to reuse code in later programs. But if you write a class instead, in a new program you can simply create new objects of that class – you've already written the class and have it sitting in a file, easily used in any new program without having to copy-and-paste or rewrite any code.

- So, those are four arguments for using classes instead of declaring all data in main and declaring all methods in the class which holds main.

- The four advantages suggest using classes is a very good idea.

- The technique of using encapsulated classes is known as *object-based programming*, since the focus is on deciding what types we need, and then writing classes to represent those types and creating objects of those types to hold our data. What we have accomplished is *data abstraction* – hiding the details of a *data implementation* away behind the scenes, and focusing only on the big picture of what the data is meant to represent conceptually. This was the same idea that we saw with *procedural abstraction* – in that case, we hid the details of how a task was performed, and focused on the big picture of what particular task those details ultimately accomplished. In our `Clock` example, we have hidden away the details of how a `Clock` is implemented, and focused on the big picture of how we would want to use a `Clock` (rather than how we would want to use integers and booleans).

```
public class Clock
{
   private int hour;
   private int minutes;

   public Clock()
   {
      this.hour = 0;
      this.minutes = 0;
   }

   public Clock(int theHour, int theMinutes, boolean theAM)
   {
      if (theAM == true)
      {
         if (theHour == 12)
            this.hour = 0;
         else
            this.hour = theHour;
      }
      else   // theAM == false
      {
         if (theHour == 12)
            this.hour = 12;
         else
            this.hour = theHour + 12;
      }

      this.minutes = theMinutes;
   }
   public void printClock()
   {
      // print variables for clock
      System.out.print("Time is ");
      if (this.hour == 0)
         System.out.print(12);
      else if ((this.hour >= 1) && (this.hour <= 12))
         System.out.print(this.hour);
      else   //   ((this.hour >= 13)  && (this.hour <= 23))
         System.out.print(this.hour - 12);
      System.out.print(":");
      if (this.minutes < 10)
         System.out.print("0");
      System.out.print(this.minutes + " ");
      if (this.hour < 12)
         System.out.println("AM.");
      else    // this.hour >= 12
         System.out.println("PM.");
   }
}
```