CS125 : Introduction to Computer Science

Lecture Notes #15
Data Composition and Abstraction: Classes and Instance Variables

Lecture 15 : Data Composition and Abstraction: Classes and Instance Variables

Representing non-primitive items

Imagine a clock. How would we represent a clock with what we know? One way is with the following set of variables:

```
int variable to hold hour (from 1 - 12)
int variable to hold minutes (from 0 - 59)
boolean to hold true if AM, false if PM
```

In the program below, we are attempting to represent two clocks. To do this, we first declare a set of variables of the above types, for each clock that we want to represent.

```
public class ClockTest
{
   public static void main(String[] args)
   {
      // declare variables for clock at home
      int homeHour;
      int homeMinutes;
      boolean homeAM;

      // declare variables for clock at office
      int officeHour;
      int officeMinutes;
      boolean officeAM;

      // main() continues on next slides...
```

Next, we can assign separate values to these separate sets of variables.

```
      // assign variables for clock at home,
      //    representing the time 2:15AM
      homeHour = 2;
      homeMinutes = 15;
      homeAM = true;


      // assign variables for clock at office,
      //    representing the time 7:14PM
      officeHour = 7;
      officeMinutes = 14;
      officeAM = false;
```

Finally, print these values out using the method below:

```
      // Prints the "home" clock
      printClock(homeHour, homeMinutes, homeAM);

      // Prints the "office" clock
      printClock(officeHour, officeMinutes,
                                    officeAM);
   }  // end main

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

}  // end of class
```

The *class* – definition of a new type

We spoke earlier of a *class* being "a helpful box to store stuff" – for example, the way the class `Keyboard` held our useful command-line input methods in one place.

However, classes have a different purpose as well, and that is to (among other things) collect variable declarations together into one. What if we take the variables we need to represent a clock, and gather them into one class?

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

If we were to do that, then `Clock` is now a new type in the language!

Some terminology

- *class* - a blueprint. The class tells us "how to make a clock", in the same way that a blueprint might tell us, "how to make a house".
- *object* - a creation whose design is described by the class, just as the house's creation is described by the blueprint. An *object* of a class, is also known as an *instance* of that class. You can create as many houses as you want from the same blueprint, and likewise, you can create as many objects as you want that all
- The variables we declared inside that `Clock` class were not declared inside any method – and thus they were not local variables, nor were they parameter variables. They are a third kind of variable, called an *instance variable*. These variables are called instance variables because every time you create a new object of a class (i.e. a new instance of that class), you also create one copy of each of the instance variables. Every time we create an object of type `Clock`, we have created a variable `hour` of type `int`, a variable `minutes` of type `int`, and a variable `AM` of type `boolean`.
- You can use a blueprint to create many houses. Likewise, use one class to create many identical objects; then customize objects with unique data values. You can use the `Clock` class above to create many different objects of type `Clock` then say, set one clock to store "2:14 AM" and another to store "7:15 PM". But each clock has two integers and a boolean.
- This is similar to building houses – every time you build a house, you have a new set of walls, a new roof, and so on. Painting the walls of one house red, won't change the color of the walls of other houses, and similarly, changing the value of `hour`, `minutes`, and `AM` in one object of type `Clock`, won't affect the values of any instance variables in any other objects of type `Clock`.

Reference variables and objects

We can declare a reference variable of type `Clock`, just as we declared a reference variable of type "`int` array". In both cases, the reference variable serves the same purpose – it will hold the address of a memory location.

Similarly, we can use the expression "`new Clock()`" to create an object of type `Clock`, just as we used the expression "`new int[6]`" to create an object of type "`int` array". Once the object of type `Clock` is created, we can use a reference variable of type `Clock` to store the address returned by the "`new Clock()`" expression – just as we had a reference of type `int[]` store address returned by the expression "`new int[6]`". In both cases, the reference variable holds the address of the memory location where the new object is stored in memory.

The type of a reference variable and the type of the object it refers to must match! The compiler will catch type-mismatch mistakes of that nature. For example, the first line below will generate a compiler error, but the second one will not:

```
int[] arr = new Clock();
Clock c1 = new Clock();
```

As with array objects, `Clock` objects – or objects of any other type (i.e. anything else created using `new`) – are *allocated* off the *heap* – a section of memory where the object does *not* go out of scope when we reach the end of the block from which it was allocated. Likewise, just as we said was the case with array objects, objects in general do not have names, and we can only access them by having reference variables refer to those objects and then working through the reference variables. All of the rules we had for reference variables and objects when we were discussing arrays likewise are true for reference variables and objects of other types.

A quick terminology issue: when we create objects of type `Clock` in this manner, we can refer to them as "`Clock` objects" (a shorter phrase to say than "objects of type `Clock`", though both phrases mean the same thing). Similarly, instead of saying "reference variable of type `Clock`", we can say "`Clock` reference variable" or "`Clock` reference". The same applies for any other type of reference variable or object.

We access the individual variables of the object by using the same dot syntax we used for arrays. Just as `length` was a variable built into every array object, likewise `hour`, `minutes`, and `AM` are variables built into every `Clock` object.

An example using what we've seen so far

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

      // create/initialize "home" clock object
      home = new Clock(); // object created
      home.hour = 2;
      home.minutes = 15;
      home.AM = true;

      // create/initialize "office" clock object
      office = new Clock(); // object created
      office.hour = 7;
      office.minutes = 14;
      office.AM = false;

      // Prints the "home" clock
      printClock(home.hour, home.minutes,
                                    home.AM);

      // Prints the "office" clock
      printClock(office.hour, office.minutes,
                                   office.AM);
   }  // end main

   // This method has not changed.
   public static void printClock(int hour,
                         int minutes, boolean AM)
   {
      // print variables for clock
      System.out.print("Time is " + hour + ":");
      if (minutes < 10)
         System.out.print("0");
      System.out.print(minutes + " ");
      if (AM == true)
         System.out.println("AM.");
      else    // AM == false
         System.out.println("PM.");
   }

}  // end of class
```

Clock references as parameters

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

      // create/initialize "home" clock object
      home = new Clock(); // object created
      home.hour = 2;
      home.minutes = 15;
      home.AM = true;

      // create/initialize "office" clock object
      office = new Clock(); // object created
      office.hour = 7;
      office.minutes = 14;
      office.AM = false;

      // Prints the "home" clock
      printClock(home);

      // Prints the "office" clock
      printClock(office);

   }  // end main

   // This method now has a Clock reference for a parameter
   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
```