Defining Your Own Classes

In C, you are allowed to define a struct and then define variables of that struct. But Java allows you to define your own class. This means not only defining the data structure, but also definition the methods (functions) that operate solely on objects of the class.

Here are the components of a class:

1) Instance Variables
2) Constructor(s)
3) Methods

The instance variables specify the different data components of an object of the class.

The constructor technically creates and initializes the object and returns a reference to it. But, really in the code that you write, all you need to do is initialize the instance variables of the object.

Each instance (non-static) method you write only gets executed if it is called with a specific object. (Remember when we called methods from the String class?) Thus, it is understood that whenever an instance variable is mentioned in an instance method, it refers to the instance variable of the object the method was called on.

In particular, each instance method provides some sort of functionality that will allow the user to manipulate objects of your class that they create.

Types of Variables in an instance method:

1) Instance Variables
2) Formal Parameters
3) Local Variables

Each of these is a different kind of variable. Even though you are allowed to, do NOT name variables of these different "types" the same name. It will cause confusion, I guarantee it. (Java has rules that specify which of the variables you are referring to if the name is ambiguous.)

As already mentioned, instance variables belong to the object the method was called on. The formal parameters serve the same purpose they do in all methods - they are the input the method needs to complete its task. Local variables also serve the same purpose in instance methods as all methods.

(Note: This handout is included on the class website.)

Here is a portion of the Time class handout:

```
public class Time {

   private int hours;
   private int minutes;

   public Time(int m) {
      hours = m/60;
      minutes = m%60;
   }

   private int totalminutes() {
      return 60*hours+minutes;
   }
   public boolean equals(Time time2) {
      if (this.totalminutes() == time2.totalminutes())
         return true;
      else
         return false;
   }

   public Time addtime(Time time2) {
      int min = totalminutes() + time2.totalminutes();
      Time temp = new Time(min);
       return temp;
   }

   public String toString() {
      return (hours+" hours and "+minutes+" minutes");
   }
}
```

The two instance variables that comprise a Time object are two integers: hours and minutes.

You'll notice that the constructor's job is simply to initialize the object. Generally the parameters of a constructor are used to initialize (set the values of) the instance variables. (Remember how this works when it is called...the constructor allocates space for the object and returns a reference to that newly created object. However, these details are hid from you even when you write the constructor! So, all you have to do is initialize the instance variables based on the formal parameters when you write the constructor.)

What NOT to do in a constructor:

1) Mistake the left&right hand sides of assignment statements.

2) Create local variables with the names of either the formal parameters OR instance variables.

A Note of Visibility Modifiers

For now, the only visibility modifiers we will use are public and private. These indicate where an instance variable or method may be accessed. In particular if something is private, it may only be referred to within the class. If something is public, it can be used anywhere.

The general rule of thumb is that all instance variables are made private. The reason is that classes allow us to create abstract data types. This means that a person can USE an object without knowing HOW it is stored or HOW the methods in the class work.

If other programmers are given access to the instance variables of the objects they create, then they have the power to manipulate the object any way they want. BUT, in order to use this power effectively, the user must understand the details of HOW the object works. THIS DEFEATS THE WHOLE PURPOSE OF FORMING CLASSES IN THE FIRST PLACE!!!

Generally, most methods are made public so that others can use them. However, it is not inconceivable to design a private method. Consider the totalminutes method in the Time class above. There is no need to allow someone USING the class to

call this method. Rather, I have simply written it so that I can carry out other methods (such as equals and addtime) with a more efficient design. Since the purpose of the method is internal to making other methods in the class, I have chosen to make it private.

Use of this

You'll notice that inside of the equals() method I have used an object called this. This can ONLY be used inside of an instance method of a class. In particular, this refers to the object the method was called on. I wrote it in this method to be explicit. If you look at the line:

```
if (this.totalminutes() == time2.totalminutes())
```

you'll see that the boolean expression is comparing the total number of minutes in the object the method is called on WITH the total number of minutes in the object time2.

Although I have been explicit here, it was not necessary to do so. Consider this line from the addtime method:

```
int min = totalminutes() + time2.totalminutes();
```

Whenever an instance method call is made without the object specified inside of another instance method, the call is AUTOMATICALLY made on the object the original method was called on, (which is this.)

There are some cases where it is necessary to use this, but usually, one can get away without using it. Standard convention is to do so - my guess is simply to save typing.

Consider the constructor rewritten with these two assignment statements:

```
this.hours = m/60;
this.minutes = m%60;
```

This (no pun intended) seems like a little overkill...

addtime Method

In this method I take in a Time object and return a Time object as well. (Notice the similarity of this prototype to that of several methods in the String class, such as concat.)

Here is the code again:

```
  public Time addtime(Time time2) {
     int min = totalminutes() + time2.totalminutes();
     Time temp = new Time(min);
      return temp;
  }
```

In order to complete the task, I first need to compute the sum of the minutes in both objects(this and time2). I do this in the first line using the totalminutes method.

Next, I need to create a Time object using the computed information. This can be done with a call to the constructor. Finally, I need to return a reference to the newly created object. (This is temp.)

As we can see here, what all the String methods that return a String must do is make a call to a String constructor inside the method and return a reference to that object that was created.

Alternate version of addtime

If we wanted addtime to automatically change the current object to the sum of the two times instead of returning a new object, we could write it as follows:

```
public void addtime(Time time2) {
    int min = totalminutes() + time2.totalminutes();
    hours = min/60;
    minutes = min%60;
}
```

Once again, we must calculate the sum of the total number of minutes in the current object and time2. But, after this calculation, rather than creating a new object, what we want to do is *change* the current object to reflect this time. We can do this by resetting both the hours and minutes components of the current object as shown above.

Thus, when this method is called (as opposed to the previous one), the object the method is called upon changes to store the sum of its "old" time and time2's time.