Defining Your Own Classes

Java allows you to define your own class. This means not only defining the data structure like the members and variables, but also defining the methods and functions that can 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; they are also referred to as members of a class.

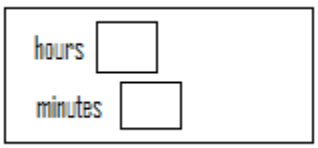
The constructor is a specific type of method that technically creates and initializes the object and returns a reference to it. But, really in the code that you write, all you need to do is initialize the instance variables of the object. Not all classes need a constructor but if you do have one, you'll generally do basic creation stuff and assignments to the members of your class, such as initializing an integer member to '0'.

A method is basically a function; it's a collection of executable lines of code and statements that generally relate to each other to accomplish a task. Instance methods only get executed if they are called with a specific object. (Remember when we called methods from the String class? If you have an object of type String named 'myString', you can call instance methods of and on that object by *myString.length()*) Thus, it is understood that whenever an instance variable is mentioned in a instance method that 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. On the other hand, static methods (like main) can be accessed and executed from the class directly without operating on an object created from that class (think of *Math.abs()*—you haven't created an object of the Math class, but you're statically accessing its method).

The easiest way to learn about how to write a class is to look at an example. Here is a portion of the Time class:

```
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 void print() {
        System.out.print(hours+":"+minutes);
    }

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

The two instance variables that comprise a Time object are two integers: hours and minutes. Thus, a picture of a Time object is as follows:



You'll notice that the constructor's job (*public Time(int m)*) is simply to initialize the object. The list of information inside the parentheses is known as the list of formal parameters to the method. For this constructor, there is one formal parameter, an integer with the name m. This means that if someone wants to use this constructor to build their Time object, then they must provide a single integer with which to build the object. Inside of the constructor, the name of that integer will be m. Thus, when we write the code for the constructor, we assume that we are ALREADY given a value stored in m, and that we are to use that value to initialize (set the values of) the instance variables (in this case, hours and minutes). In this instance (sorry for the pun), we want to set hours to m/60, since this represents the number of full hours in a time period of m minutes, and we want to set minutes to m%60, since this represents the number of minutes leftover after we've accounted for all the whole hours.

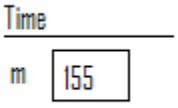
Remember how this works when it is called...the constructor allocates space for the object and returns a reference to that newly created object. However, these details are hid from you even when you write the constructor! So, all you have to do when writing it is initialize the instance variables based on the formal parameters, as we've done here.

What NOT to do in a constructor:
1) Mistake the left- and right-hand sides of assignment statements.
2) Create local variables with the names of either the formal parameters OR instance variables.

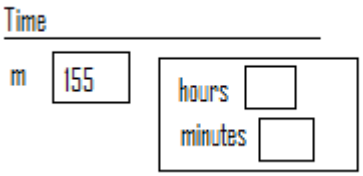
Just so we can see how this interacts with someone CALLING the Time constructor, imagine that we're in the main method (of any class) and we have the following line of code

```
Time t = new Time(155);
```

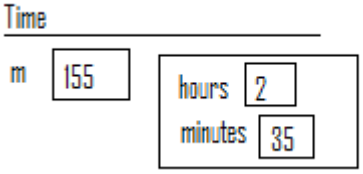
When the constructor is called, the very first step is to set the formal parameter, m, to the value put in that corresponding spot, which is 155. Thus, when the code for the constructor starts running its memory simply has one slot for a variable m which stores 155:



Next, before we start any of the code, the constructor finds memory for the object, so our picture becomes:



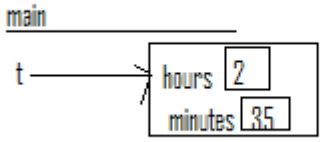
Now we start executing the lines of code. The first line will set hours (which automatically refers to the hours inside the object being created) to 155/60, which is 2. The second line of code sets minutes to 155%60, which is 35. Right before the constructor is done, the picture is as follows:



Finally, the constructor's last task is to return a reference to where the object it created is. When we go back to main, the line of code was:

```
Time t = new Time(155);
```

Thus, t is a reference defined in main and it will refer to the object that the constructor just created, since this is what the constructor returns, a reference to the newly created object. The picture, from main's point of view is as follows:



Notice that our picture no longer includes memory for the Time constructor or its formal parameter m. This memory was wiped out when the constructor finished.

Void Methods

A void method is one where the return type designated is void. This means that the method, once it's done running, has no information to return to who called it. Usually, void methods carry out some specified task and then complete. An example of a void method is the print method in the Time class:

```
public void print() {
    System.out.print(hours+":"+minutes);
}
```

The job of this method is to print out the time object. Once this task is done, nothing needs to be returned. This method takes in no formal parameters because it needs no new information to do its task. All the information it needs (the number of hours and minutes in the object) is embedded in the instance variables of the object, to which this method has access. Thus, inside a void method, you can write as many lines of code as you want, just like a little mini-program. This particular mini-program just prints out the time object.

To see how we would call it from main, imagine calling the method right after we created the time object t with 155 minutes:

```
t.print();
```

Notice that in order to call this method, we MUST call it ON a Time object, in this case, t. (Technically, t is a reference that refers to the time object we just created…)

Now, when print executes, it accesses the hours and minutes components of the object t is referring to, and it prints out:

```
2:35
```

Methods that Return Values

An example of a method that returns a value is the totalmintes method:

```
private int totalminutes() {
    return 60*hours+minutes;
}
```

For now, don't worry about private, which is a visibility modifier. This will be cleared up shortly. A method that returns a value has some sort of task and then must return some information to the method that called it. For example, whoever calls the totalminutes method wants to know how many total minutes are in the current time object. It's the method's job to calculate this value and return it! Thus, we have a new type of statement called a return statement. Whenever we know what we want to return inside of the method, we start the statement with the word return, followed by a space, followed by an expression that has the value we wish to return. (This is of course, followed with a semicolon.) When a return statement is executed, the computer immediately transfers control back to the method that called it. Thus, if the addTime method calls totalminutes, then control returns back to addTime right after the return statement in totalminutes executes. In this method, we have access to the object and simply calculate 60*hours + minutes, and return it.

In the section that goes over the addTime method, a trace through will be done to show the interaction between the addTime method and the totalminutes method.

Parameters in Method

We can view parameters from two places:

(1) Method call
(2) Inside the method itself

Parameters from a method call are called "actual parameters."

In the lines of code

```
Time t = new Time(155);
double a = 1, b = -8, c = 12;
double x = (-b + Math.sqrt(b*b - 4*a*c))/(2*a);
```

we have method calls to the Time constructor and the sqrt method. The actual parameter to the first method call is 155 and to the second is b*b – 4*a*c.

Notice that actual parameters, if they of a primitive type (int, char, double, etc.), can be any expression that evaluates to the appropriate type.

If we have a the following situation:

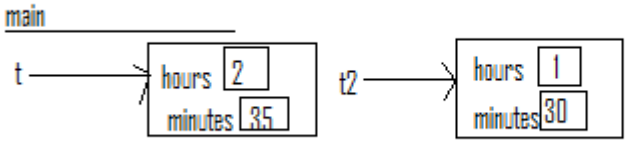
```
Time t = new Time(155);
Time t2 = new Time(90);
Time t3 = t.addTime(t2);
```

We see that the actual parameter to the addTime method is t2. In this situation, since the parameter type is a non-primitive, the actual parameter in its place MUST BE a reference to an object of the appropriate type.

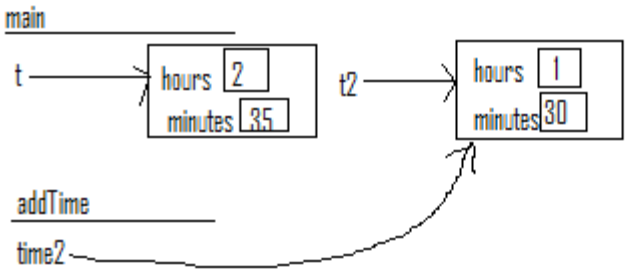
Parameters in the method definitions are known as formal parameters. The name of the formal parameter to the Time constructor shown here is m and its type is int. The name of the formal parameter to the addTime method is time2.

A picture has already been shown showing how an actual parameter is passed to a formal parameter for a primitive value. (Namely, the value of the actual parameter is calculated and this value is the copied into the appropriate formal parameter slot.)

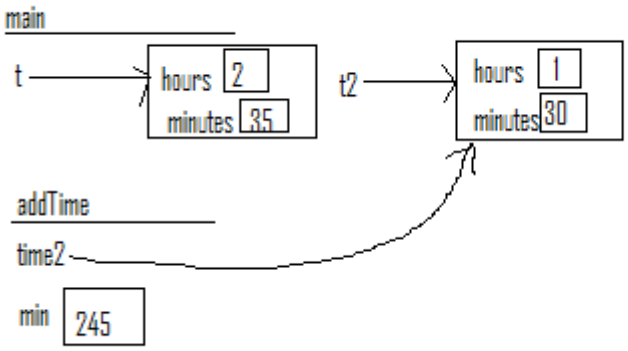
Now, let's do a quick trace through of the mechanics of passing a parameter that is a reference (instead of a primitive). Consider the three lines of code shown above. Assume they are in a main method. After the first two lines of code, the picture looks as follows:



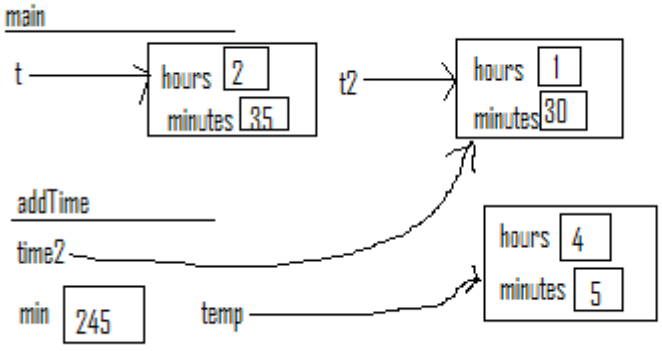
Now, when addTime gets called from main, it gets its own separate memory. It also gets a slot for its formal parameter. The very first thing that happens is that the formal parameter will point to the same object that the corresponding actual parameter reference points to. Thus, at the beginning of the addTime method call, our picture looks like the following:



Next, from inside the addTime method, create a local variable min and then we call the totalminutes method, once on the current object, which is t, and then we call it on t2. The first time we call it 155 is returned. The second time we call it 90 is returned. We add these two values to obtain 245. This is then stored in the local variable min:



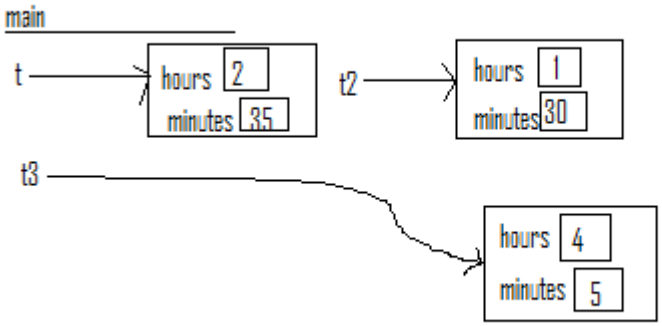
Then we create a local reference called temp, call the constructor for Time with 245 and make temp point to this newly created object. Since we've previously traced through the steps of the Time constructor, we'll omit them here and just show the end result:



In the last line of code in the addTime method, we return temp. This means we are returning a reference to where the object that temp is referring to is. In main, the line of code that receives this return is

```
Time t3 = t.addTime(t2);
```

Thus, t3 will be a new Time reference in main and will point to the same object that temp from addTime pointed to:



A few things to notice here:

(1) All the memory for the method addTime is gone.
(2) The memory for actual objects doesn't reside in any method, thus, the object created by addTime persists even after the method is done. BUT, the references inside of addTime (temp) cease to exist after the method call is done.
(3) The task of local variables is to simply aid the method in completing its task. Once the method is done, they are no longer needed.
(4) The task of actual parameters is to give a method the information it needs to solve the problem at hand.
(5) The task of formal parameters is to receive the information given by the actual parameters. If the parameter type is a primitive, the value of the actual parameter is COPIED into the corresponding formal parameter, which then acts as a local variable in the method. If the parameter type is a reference, then the formal parameter will refer to the same EXACT object that the actual parameter does. Though this example didn't show it, the method addTime COULD have changed the contents of the object that t2 in main was referring to.

Types of Variables in an instance method

Let's review the three types of variables that one is allowed to use in an instance method:

1) Instance Variables—members of the class
2) Formal Parameters—arguments passed to the method
3) Local Variables—"temporary" variables used only within the method itself

These are three different kinds of variables. 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 as instance variables but are inaccessible outside of the method.

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. For instance, the Time class's minutes variable is private, so if you created a Time object called 'myTime', you could not directly access its minutes by *myTime.minutes*. If it were public, you could do that. 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!!!

Sometimes, particular private variables have publically-accessible getter and setter methods created for them; these are essentially nothing more than public methods that either return or set the value of the private variables. Of course, there might be particular ways these variables need to be changed, which is why the class author restricts the ways in which the variables contents can be changed or read.

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 what an object called *this*. *this* can ONLY be used inside of a non-static 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 a non-static method call is made without the object specified inside of another non-static 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 a bit like overkill...