COP 3330 – Object Oriented Programming in Java
Creating Your Own Classes (finally, Objects!)

Creating Your Own Type AND Associated Methods

When Java's primitive types and built in classes (such as String) aren't adequate to store the aggregate the data you need to, Java allows the programmer to define their own class.

In C, no such thing exists, but one can define a struct (essentially your own type), and then write functions that take in the struct as a parameter. The issue with this paradigm is that any programmer calling the function must understand how the struct is built and roughly how it works.

In order to enhance the power of programming, one key tool is data abstraction. If a programmer doesn't have to understand how the data is stored in an object, but just needs to know the behaviors that are allowed "on the object", then the programmer can very quickly learn to use objects as they need without fully understanding how they are stored or how the associated methods that operate on that object work. This is true of regular functions: you may call a function and use its result without fully understanding how it works (think some of the math functions like acos...). But, a deeper level of abstraction is data abstraction, not needing to understand how the data is stored. Only object-oriented programming can provide data abstraction. In our initial examples, we won't see this as a self-evident fact, but later in the semester, we'll see clear examples of data abstraction.

Thus, to define your own class, you not only have to define the components that store the data, but also define methods that operate solely on objects of the class. We can think of a class as a blueprint and an object as a specific instantiation of that blueprint. (So class is similar to but not identical to type and object is similar to a variable identifier.)

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. Before we dig into a concrete example, let's look at what variables we are allowed to use inside of an instance method, as this is perhaps more complicated than static methods.

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.)

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. You define them temporarily for some subtask (think loop index i).

It is common for students learning to confuse instance variables with formal parameters. **Try really hard to immediately distinguish between the two and don't move on until you are 100% sure which variables are of which type.**

For the remainder of this lecture, we'll look at three user defined classes:

1. Time_V1 Class
2. MagicEightBall class
3. MagicEightBall_V2 class

Our Time class will store time in hours, minutes and seconds. We will allow the following operations for our Time objects:

1) Adding two Time objects.
2) Increasing the value of a Time object.
3) Subtracting two Time objects.
4) Decreasing the value of a Time object.
5) Comparing two Time objects.

The general layout of a class should look like this:

```
public class ClassName {

    // List constants, static variables.

    // List instance variables.

    // List constructors.

    // List methods.
}
```

Here is a portion of the Time_V1 class with the constants, instance variables and constructors:

```
public class Time_V1 {

    // Useful classes.
    final public static int SEC_PER_MIN = 60;
    final public static int MIN_PER_HR = 60;
    final public static int SEC_PER_HR = SEC_PER_MIN*MIN_PER_HR;

    private int hours;  // number of hours in the time object.
    private int minutes; // number of minutes in the time object.
    private int seconds; // number of seconds in the time object.

    // We assume h, m and s are all non-negative.
    // A constructor that takes in both the number of hours and minutes for
    // the time object to be constructed. The resulting object is created
    // such that the number of minutes and seconds is in between 0 and 59.
    public Time_V1(int h, int m, int s) {
        int total = SEC_PER_HR*h + SEC_PER_MIN*m + s;
        hours = total/SEC_PER_HR;
        minutes = (total%SEC_PER_HR)/MIN_PER_HR;
        seconds = total%SEC_PER_MIN;
    }

    // This constructor takes the total number of seconds for the Time object.
    public Time_V1(int totalSec) {
        hours = totalSec/SEC_PER_HR;
        minutes = (totalSec%SEC_PER_HR)/MIN_PER_HR;
        seconds = totalSec%SEC_PER_MIN;
    }

    // Default constructor, sets the time object to 0 hours and minutes.
    public Time_V1() {
        hours = 0;
        minutes = 0;
        seconds = 0;
    }

    // Rest of the methods...
}
```

The top has three constants that will help us with conversions between hours, minutes and seconds, as needed. This is followed by the three instance variables for a Time_V1 object, hours, minutes and seconds, each stored in an integer. The syntax here is to list the visibility modifier, the type and the instance variable name, separated by white space. Typically, we'll make instance variables private, so that those outside the class can not access them. Private means that these variables can only be accessed within the class.

Then, we provide three constructors. Java requires us to provide at least one constructor for an object and it's normal to have more than one. This is called method overloading. Java knows which method is being called because the parameter lists to overloaded methods must be different in number of parameters or at least one corresponding type.

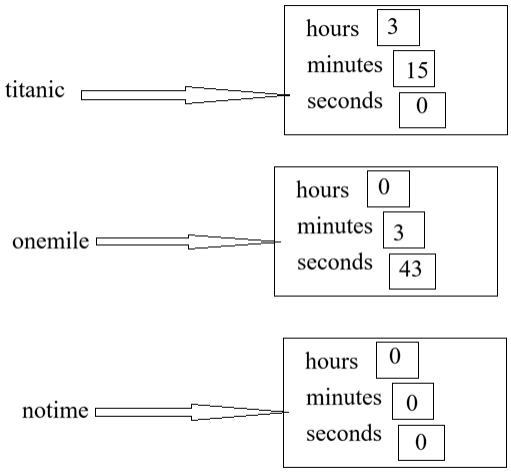
The constructor syntax is special. You always use the visibility modifier public, followed by the class name, then the parameter list. Inside the method, the goal of the constructor is to build the object. This basically just means to set its instance variables to the appropriate values, based on the formal parameters. Let's breakdown the first constructor:

```
public Time_V1(int h, int m, int s) {
    int total = SEC_PER_HR*h + SEC_PER_MIN*m + s;
    hours = total/SEC_PER_HR;
    minutes = (total%SEC_PER_HR)/MIN_PER_HR;
    seconds = total%SEC_PER_MIN;
}
```

To call this constructor, the user has to pass in the number of hours, followed by the number of minutes, followed by the number of seconds. **The formal parameters to this method are h, m and s, respectively. The instance variables are hours, minutes and seconds.** In our code, we immediately calculate the total number of seconds, then use modulo math to properly assign the number of hours, minutes and seconds in our object. It's key to understand that hours, minutes and seconds must be on the left hand side of the assignment statements and h, m and s, should not be. Also, do NOT create local variables with the same name as either of the formal parameters or instance variables. (In the code above, total is the only local variable.) Here are three method calls to these three constructors:

```
Time_V1 titanic = new Time_V1(3, 15, 0);
Time_V1 onemile = new Time_V1(223);
Time_V1 notime = new Time_V1();
```

So, outside of the class (or maybe in the main method of the Time_V1 class), The syntax of creating a Time_V1 object is to use the class name, followed by the reference name. Then, set the reference equal to what the constructor returns. To call the constructor, use the keyword new followed by a call to the constructor, which means just using the classname and parentheses, passing in the appropriate parameters. Here is the picture that corresponds to those three lines of code:



One key method we will use inside of the Time_V1 class but that we don't want to make public to those who just use the class (and don't develop it) is the totalSeconds method:

```
private int totalSeconds() {
    return SEC_PER_HR*hours + SEC_PER_MIN*minutes + seconds;
}
```

For the user to get good use of their Time_V1 objects, they don't need to use this method, but it's very helpful to us in writing the public methods we do want the user to be able to use. This is why this method is private.

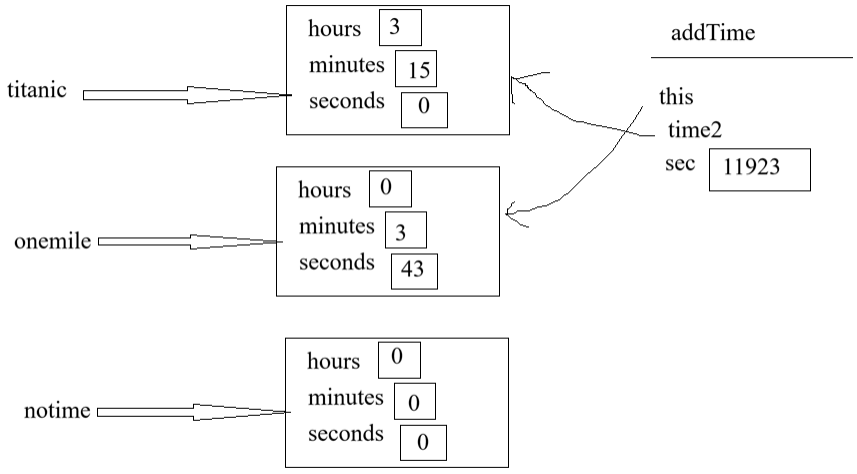
Now, let's take a look at one method that doesn't mutate an object and returns a new one (like the String class:

```
public Time_V1 addTime(Time_V1 time2) {
    int sec = totalSeconds() + time2.totalSeconds();
    Time_V1 temp = new Time_V1(sec);
    return temp;
}
```

This method takes in a reference to a Time_V1 object and returns a new Time_V1 object. Because it's an instance variable, it has to be called upon a Time_V1 object. Thus, there are three different Time_V1 references in play when this method is called (instantiated). Note that as previously discussed, the way to call an instance method is to use the object.methodname syntax as we do with time2.totalSeconds(). If we don't explicitly write out the object upon which an instance method is called, then the object is **this.** Consider the following method call:

```
Time_V1 newtime = onemile.addTime(titanic);
```

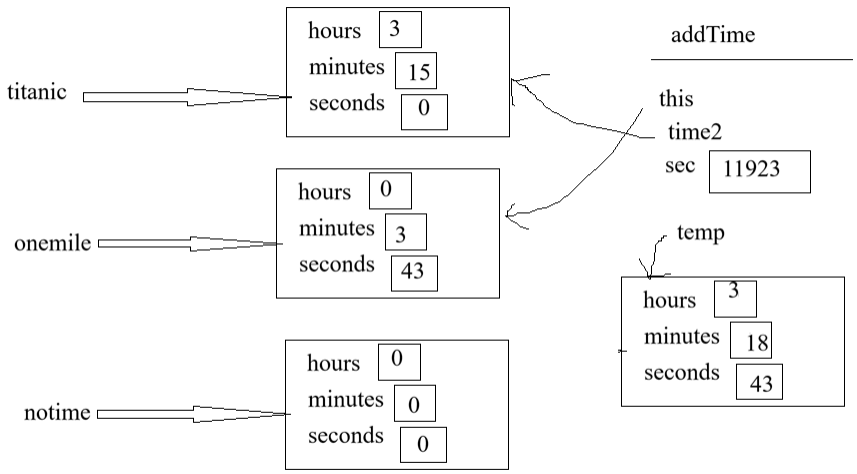
The addTime method is attached to the object referenced by onemile. So, right when the method is called, here is our picture. Since the method has to be called on an object, Java gives a special name within the instance method to the object the method is called on: this. You will see this in the picture in this way. (Sorry, pun intended.) Here is the picture after the first line of the method:



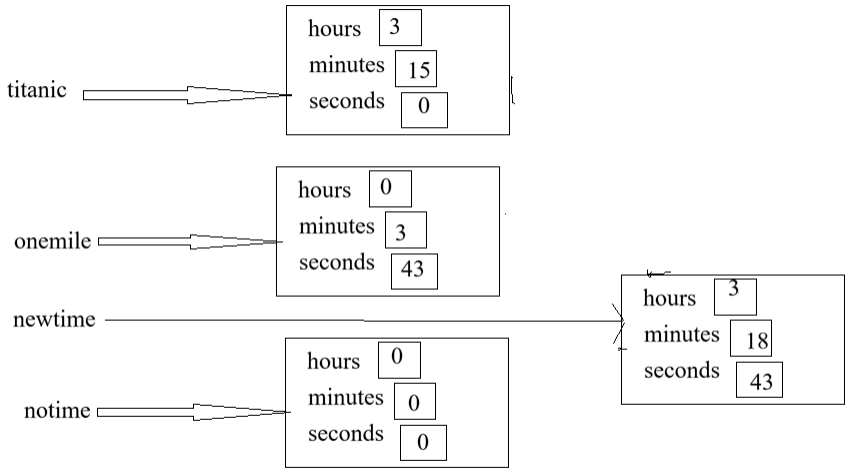
The next line of code in the method creates a new Time_V1 object:

```
Time_V1 temp = new Time_V1(sec);
```

Here's what the picture looks like after this line:



When temp gets returned, the memory for the addTime method is gone and in main the reference newtime will reference the newly created object:



This is essentially how all the String methods work. None of them change any existing object. Instead, many of them return new objects related to the objects that the String methods were called upon. Now, let's take a look at a two other methods in the Time_V1 class. One is private for our internal use and the second is similar to the one above, but mutates a Time_V1 object.

```
private void updateTime(int newTotalSec) {
    hours = newTotalSec/SEC_PER_HR;
    minutes = (newTotalSec%SEC_PER_HR)/MIN_PER_HR;
    seconds = newTotalSec%SEC_PER_MIN;
}
```

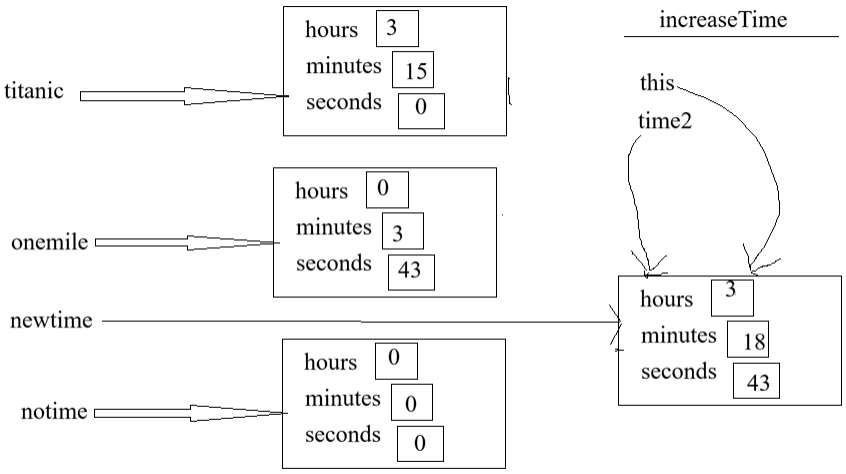
This method allows us to update all of the instance variables in a Time_V1 object according to the total number of seconds, newTotalSec. We'll use this method in the following public method that mutates a Time_V1 object:

```
public void increaseTime(Time_V1 time2) {
    updateTime(totalSeconds() + time2.totalSeconds());
}
```

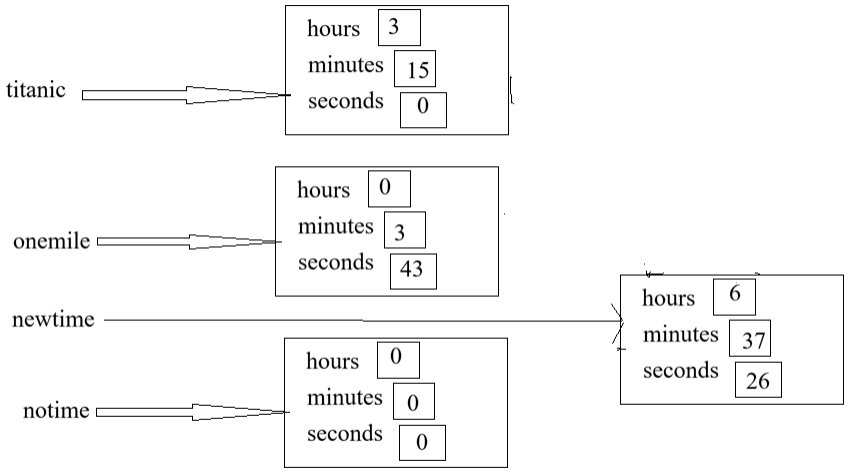
Let's consider the following method call:

```
newtime.increaseTime(newtime);
```

This is the picture right when the method call is made:



Both totalSeconds calls will return 11,923 since the two different references are referencing the same object! (See that over time the same reference can point to different objects as well, though we haven't seen that example yet, and here multiple references can point to the same object also.) The corresponding sum is 23,846. Now, when updateTime is called, it's called on the object referenced by this, so this updateTime method will change the hours to 23846/3600 = 6, then change the minutes to (23846%3600)/60 = 37, and the seconds to 23846%60 = 26. Since this is referring to the object on the right in the picture, when the method ends (and it doesn't return anything), our picture is:



There is a separate program written in a different class that uses the Time_V1 class called, PuzzleSolver.java. This program starts by asking the user to enter a time limit to solve several puzzles. Then, it loops asking the user how much time it took to solve each subsequent puzzle. If the time is less than equal to the time left on the clock, then the time is subtracted and the puzzle is counted as solved. Otherwise, the puzzle solving ends without solving the last puzzle. At the end of the program, the total number of puzzles solved is printed.

This is a typical application that uses a class. It doesn't call most of the methods in the class; just the ones it needs. But the class has to provide methods for all possible users who might be using the class for very different purposes. In this particular application, the key method used is the decreaseTime method shown below:

```
public boolean decreaseTime(Time_V1 time2) {

    int totalSec = totalSeconds() - time2.totalSeconds();
    if (totalSec < 0) return false;

    updateTime(totalSec);
    return true;
}
```

This method will decrease the time of a time object (similar to increaseTime but in the opposite case direction), so long as the formal parameter is referencing an object that has less than or equal time to this time object. If this isn't the case, false is returned and no change is made to this object. Alternatively, the time object is decreased by the amount of time in the object referenced by time2 and true is returned.

In our application, we ask the user for each puzzle how much time it takes to solve and then try to decrease the time of our overall timer. Before the main loop, we do the following to set our timer:

```
Time_V1 timeLeft = new Time_V1(hr, min, sec);
```

Here, the local variables hr, min and sec store the values entered by the user as being the total time they have to solve puzzles.

Then, inside of the main loop after the user enters how long a puzzle will take, we do the following:

```
Time_V1 toSub = new Time_V1(hr, min, sec);
boolean tmp = timeLeft.decreaseTime(toSub);
```

Basically, we create a new time object based on what the user enters for the time to solve the current puzzle and try to decrease our timer (timeLeft) by the amount of time the current puzzle took (toSub). If this succeeds, we have solved another puzzle. If not, we end our quest to solve puzzles. The entire program is found in the Sample Programs file PuzzleSolver.java. Notice that so long as both PuzzleSolver.java and Time_V1.java are in the same directory and we compile PuzzleSolver.java, Time_V1.java naturally gets compiled and the Java interpreter will be able to see all of the instance methods in the Time_V1 class from the PuzzleSolver class.

A Note on 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 totalSeconds 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.

MagicEightBall class

A Magic Eight Ball is a toy that kids use to ask questions about the future. Typically, you think of a yes/no question, shake the ball and one of several responses comes up as an answer to your question. We can model this as an object that has several possible Strings as responses and each time the user wants a response, a randomly selected String is returned. (So this object is very basic. You can build it and you can get a response from it. That's it!)

In this first version of the class, there are three possible responses the user could get. The user can custom build these responses if they want, or use the default choices provided by the default constructor. A default constructor is one that takes nothing in. (In the Time_V1 class, this was the constructor that just set the object to 0 time.)

Here is the whole class except the main method:

```
public class MagicEightBall {

    private String response1;
    private String response2;
    private String response3;
    private Random r;

    // Default object.
    public MagicEightBall() {
        response1 = "YES";
        response2 = "NO";
        response3 = "MAYBE";
        r = new Random();
    }

    // Build our object to set each of our repsonses.
    public MagicEightBall(String s1, String s2, String s3) {
        response1 = s1;
        response2 = s2;
        response3 = s3;
        r = new Random();
    }

    // Returns one of the possible responses via random choice.
    public String getResponse() {

        // Choose a random number (0,1,2)
        int choice = r.nextInt(3);

        // Return the appropriate choice.
        if (choice == 0)
            return response1;
        else if (choice == 1)
            return response2;

        return response3;
    }
}
```

Here is a basic way to use the class:

```
MagicEightBall mine = new MagicEightBall();
for (int i=0; i<10; i++) {
    System.out.println("Think of a yes/no question.");
    System.out.println(mine.getResponse());
}
```

You can customize the object by passing in the three possible responses that you want the MagicEightBall to be able to answer into the constructor.

A natural way to extend this class is to have the possible answers stored in an array. That is what the MagicEightBall_V2 class does. This even simplifies the getResponse method by removing the loop! Here's the class:

```
public class MagicEightBall_V2 {

    // Our instance variables - all our responses and the random object.
    private String[] responses;
    private Random r;

    // Default object.
    public MagicEightBall_V2() {
        responses = new String[3];
        responses[0] = "YES";
        responses[1] = "NO";
        responses[2] = "MAYBE";
        r = new Random();
    }

    // Build our object to set each of our repsonses.
    public MagicEightBall_V2(String[] options) {
        responses = options;
        r = new Random();
    }

    // Returns one of the possible responses via random choice.
    public String getResponse() {
        return responses[r.nextInt(responses.length)];
    }
}
```