More on Classes

toString

One “special” method in Java is the toString method. The method (regardless of which class it’s added to) has the following prototype:

```
public String toString();
```

The job of this method is to return a String representation of the object. Here is the toString method from the Time class:

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

In particular, let’s say we define a Time object somewhere as follows:

```
Time t = new Time(75);
```

If we follow this statement with

```
System.out.println(t);
```

One’s initial reaction might be, “how can we know how to print out a Time object?” The answer to this question is that whenever Java sees this, it automatically converts this line of code (without the user specifying) to the following:

```
System.out.println(t.toString());
```

Thus, in the context of a print or println statement, Java knows to replace an object with its String representation, which is defined by the toString method.

This begs the question, “what if I don’t write a toString method in a class?” Every class the user creates (and every class in Java’s API) inherits from the Object class automatically. This object class has a definition for the toString method. (We’ll talk about inheritance later.) Thus, what happens if you forget to define a toString method is that the Object class’s toString method gets called. This usually prints something like the following:

```
Time@1ef723
```

The first part of this output is the name of the class of the object. This is followed by the ‘@’ character. The last six digits are hexadecimal digits (don’t worry about what this means for now) that represent a location in memory where this Object is located.

Clearly, this isn’t what we want to print out. Thus, to avoid this, you should define a toString method in every class you write.

Comparable Interface

An interface in Java is essentially a contract to write a set of methods. One of the most common interfaces in Java to implement is the Comparable interface. In order to implement this interface, you must write a method with the following prototype in your class (if you were creating the Time class):

```
public int compareTo(Time time2);
```

Obviously, if you were creating a class other than the Time class, this is what the type of the input parameter would be. The job of this method is as follows:

The method should return a negative integer if the current object (this) comes BEFORE t. The method should return 0 if the two objects (this and t) are equal. The method should return a positive integer if this object comes AFTER t.

Thus, this interface should ONLY be implemented if two different objects from the class can be compared in order. This should be fairly easy to add to our Time class:

```
public int compareTo(Time time2) {
    return this.totalminutes() – t.totalminutes();
}
```

Notice how the mathematical expression listed above corresponds very nicely to the requirements of the compareTo method. Often times, an series of if statements is used to implement a compareTo method, but in this particular case, the mathematics works out nicely so that an if statement can be avoided.

In order to truly implement the interface, this final change has to be made at the beginning of the class:

```
public class Time implements Comparable<Time> {
```

We won't utilize the Comparable interface now, but once we create arrays of objects, creating a class that implements Comparable will allow us to use Java's built-in sort methods. (These come in very handy!)

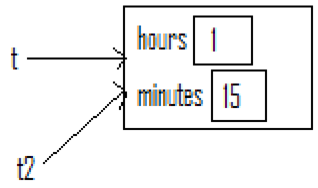
References versus Objects

In the pictures in the previous set of notes, a distinction was drawn between references and objects. References have names and objects don't. Thus, in those examples, t, t2, and t3 were all examples we have seen so far, the number of references available has equaled the number of objects. Because this is frequently the case, students often confuse references and objects as being one in the same.

The following example will illustrate the difference between the two:

```
Time t = new Time(75);
Time t2 = t;
```

After these two lines of code, the picture is as follows:



The key difference here is that there is only one object, but there are two references. The second line of code makes the reference t2 point to the same object that t points to.

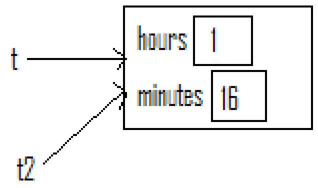
Now, imagine adding the following method to the Time class:

```
public void addMinute() {
    minutes += 1;
    if (minutes > 59) {
        minutes = 0;
        hours++;
    }
}
```

This method ***changes*** the object upon which it is called. Thus, if we made the following call:

```
t2.addMinute();
```

Our new picture would be as follows:



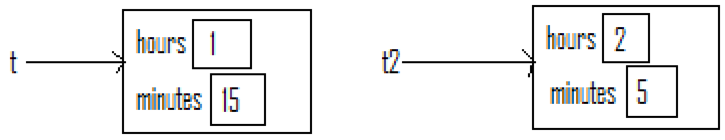
It's clearly evident from this picture that even though we didn't call any method on t, if we were to print out t, it would print out as "1 hours and 16 minutes".

Practical use of an Extra Reference

In certain instances, having more than one reference to the same object is what we want. Consider the situation where we want to swap two Time objects:

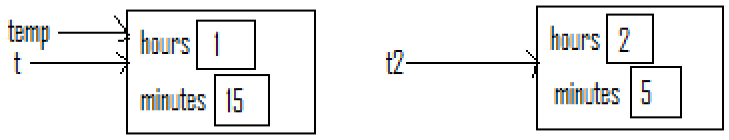
```
Time t = new Time(75);
Time t2 = new Time(125);
```

The picture after these two lines of code is:



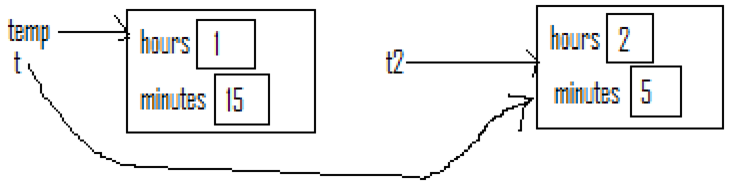
Let's say we want t to refer to the Time object that is 2 hours and 5 minutes and we want t2 to refer to the Time object that is 1 hour and 15 minutes. The truth is that we don't REALLY need a third Time object. We do, however, need a third Time reference:

```
Time temp = t;
```



Now, we are free to move the reference t:

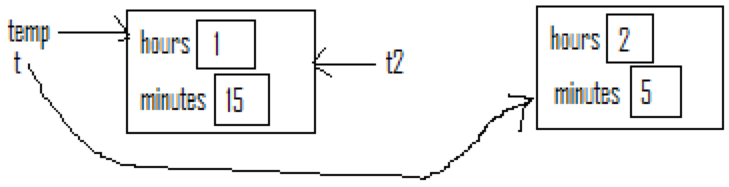
```
t = t2;
```



Notice how this line of code just MOVES the reference t to point to the same object that t2 points to, as was previously discussed. No new object is created here, but rather, a reference has moved.

Now, we conclude with the line:

```
t2 = temp;
```



The swap is complete and we never had to create a third object!

Cloning

Other times, however, we will want to create a real copy of an object, not just have two separate references pointing to the same object. The way to do this is through what is often called a "copy constructor." (This terminology is usually only used in describing the same function in the language C++.) This simply means creating a constructor that takes in an object from the class. Here is a copy constructor for the Time class:

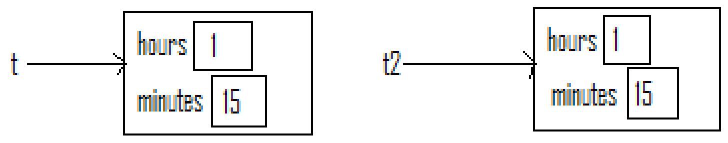
```
public Time(Time time2) {
    hours = time2.hours;
    minutes = time2.minutes;
}
```

It should be fairly obvious that what this constructor does is take in a Time object and simply copy each of its components into the new object being created by the constructor.

Now, consider these two lines of code:

```
Time t = new Time(75);
Time t2 = new Time(t);
```

The corresponding picture is as follows:



Here, t2 is NOT referring to the same object as t.

Method Overloading

Method overloading is the practice of creating two methods in the same class with the same name. In order to be allowed to do this, the two methods must have parameter lists that are different. This means that at least one corresponding type for parameters must be different. Obviously, any two methods with a different number of parameters can automatically be distinguished from one another.

The compiler decides which method of multiple methods of the same name gets called by looking at the types of the actual parameters. These types must match the corresponding formal parameters.

Consider the overloading of the add method in the fraction class illustrated below:

```
public fraction add(fraction f) {

  int num = this.num*f.den + this.den*f.num;
  int den = this.den*f.den;

  fraction temp;
  temp = new fraction(num, den);
  return temp;
}

public fraction add(int n) {

  int num = this.num + this.den*f.num;

  fraction temp;
  temp = new fraction(num, this.den);
  return temp;
}
```

We can tell which of these two methods to call based upon whether the user passes us a fraction or an integer as the only parameter. It makes sense to name both of these add because they both add things. The first adds a fraction to this fraction while the other adds an integer to this fraction.

Static Methods

The term static means, “belongs to the class.” Thus, a method that is static does NOT pertain to an object. All the methods in the Math class are static because they do NOT operate on, or pertain to, a Math object.

In the Time class, each method does pertain to a Time object. But, there are other situations where a method might be included in a class, even though it doesn’t pertain to that object. For example, consider creating a fraction class. It would have to instance variables, one to store the numerator and another to store the denominator. Usually, when dealing with fractions, it’s good to keep the fraction in lowest terms. In order to internally do this, we would have to “reduce” fractions after calculations. This reduction involves finding the greatest common divisor between the numerator and denominator. Typically speaking, the task of finding a greatest common divisor between two integers is one that isn’t tied to any object. Thus, it would make sense in the fraction class to have a method with the following prototype:

```
public static int gcd(int a, int b);
```

The rules for writing a static method are nearly the same as writing a regular instance method. The ONLY difference is that this method does NOT operate on any object. Thus, you do NOT have access to ANY instance variables inside of this method. Furthermore, you CAN’T call an instance method from a static method, UNLESS you specify which object you are calling the instance method on. The rules for how the formal and actual parameters work, parameter passing and return values are identical to instance methods.

If you think to all of your programs you wrote BEFORE we introduced classes, each of these programs had no objects. But, there is no reason these programs couldn’t have been split up using multiple static methods.

The following example that carries out a blackjack game between the user and the computer is a fairly detailed example that utilizes 5 static methods. (One of those methods is main…) Here are each of the method prototypes (except main) with a brief description of each:

```
// This method executes the game for the user/player.
public static int playPlayer(Scanner stdin, Random r);

// This method executes the game for the computer.
public static int playComputer(Random r);

// Given both the player's score and the computer's score, this
// method prints out who won the game.
public static void printWinner(int playerscore,
                               int computerscore);

// This method retrieves the score of an Ace for the user by
// asking him/her what value they would like their ace to be.
public static int getAce(Scanner stdin);
```

```
/**********************************************************************
*                          Black Jack program                        *
*                              Arup Guha                             *
*                               2/8/99                               *
*                                                                    *
*  Brief Desciption : This program lets the user play one had of     *
*  blackjack against the computer. The main difference beteweent his *
*  implementation versus normal blackjack is that the user must decide *
*  immediately if an ace is to be worth 1 point or 11.               *
*                                                                    *
*  Edited on 7/20/05 for 2005 BHCSI: Main change made was that the old *
*  code didn't have any methods. This code has been streamlined with *
*  the use of several methods.                                       *
**********************************************************************/

// Edited on 7/22/07 to utilize the Scanner for I/O.

import java.util.*;
import java.lang.Math;

class Blackjack {

    public static void main(String argv[]) {

            Scanner stdin = new Scanner(System.in);

            Random generator = new Random(); // used to generate rand cards.

            int score, computerscore; // keeps track of scores

            System.out.println("Welcome to Blackjack against the computer.");

            // Prints out player's final score
            score = playPlayer(stdin, generator);
            System.out.println("So, your final score = " + score);

            // Prints out computer's final score.
            computerscore = playComputer(generator);
            System.out.println("The computer score = " + computerscore);

            printWinner(score, computerscore);
    }

    // This method executes the game for the user/player.
    public static int playPlayer(Scanner stdin, Random r) {

            boolean hit = true; // true while player wants to hit.
            int numcards = 0; // keeps track of the # of cards the user has.
            int score = 0;

            // Loop runs till user doesn't want another card or has busted.
            while (hit) {

                int singlecard = 0; // Makes sure following loop runs
                                    // only 1 after initial two cards are given.

                // Gives player 2 cards initially, then 1 a time.
```

```
        while (numcards < 2 || singlecard < 1) {

          int rand = Math.abs(r.nextInt())%13; // next card.

          // Ace case: Let's user choose value.
          if (rand == 0) {
              score = score + getAce(stdin);
          }

          // Scores other cases.
          else if (rand < 10) {
              System.out.println("Your card value = " + (rand + 1));
              score = score + rand + 1;
          }
          else {
              System.out.println("Your card value = 10");
              score = score + 10;
          }

          // Increment appropriate counters.
          numcards++;
          singlecard++;
        }

        // Prints out player's current score and asks
        // if he/she wants another hit.
        System.out.println("So, your current score is = " + score);
        if (score <= 21) {

                System.out.println("Would you like to hit again?");
                char ans = (stdin.next()).charAt(0);

                if (ans != 'y' && ans != 'Y')
                hit = false;
        }
        else
                hit = false;

    } // while hit

    return score;
}

// This method executes the game for the computer.
public static int playComputer(Random r) {

    int computerscore = 0;

    // Executes computers turn. It hits till 17, just like Vegas.
    while (computerscore < 17) {

      int rand = Math.abs(r.nextInt())%13; //next random card.
      int cardvalue; // used to store current card's point value.

      // Picks score of an ace, so as not to bust.
      if (rand == 0) {
```

```
                        if (computerscore < 11)
                          cardvalue = 11;
                        else
                          cardvalue = 1;
                }
                else if (rand < 10)
                        cardvalue = rand + 1;
                else
                        cardvalue = 10;

                // Prints out computer's score after every card.
                computerscore = computerscore + cardvalue;
                System.out.println("Computer card value = " + cardvalue);
            }
            return computerscore;
    }

    // Given both the player's score and the computer's score, this method
    // prints out who won the game.
    public static void printWinner(int playerscore, int computerscore) {

            // Prints out winner to the screen.
            if (playerscore > 21 && computerscore > 21)
              System.out.println("We have both busted. No winners today.");
            else if (playerscore > 21)
              System.out.println("You have busted. Ha ha, I win!!!");
            else if (computerscore > 21)
              System.out.println("I have busted. You win by default.");
            else {

              if (playerscore > computerscore)
                System.out.println("You won!!!");
              else if (playerscore < computerscore)
                System.out.println("A computer beat you at blackjack.");
              else
                System.out.println("We have tied.");
            }

    }

    // This method retrieves the score of an Ace for the user by asking
    // him/her what value they would like their ace to be.
    public static int getAce(Scanner stdin) {

            System.out.println("You have gotten an ace, 1 or 11?");
            int acescore = stdin.nextInt();

            // Continue prompting for a score until you get a valid one.
            while (acescore != 11 && acescore != 1) {

              System.out.println("Not valid, please try again.");
              acescore = stdin.nextInt();
            }
            return acescore;
    }

}
```