Carleton University
COMP1406/1006, Fall 2013
Tutorial 3

Week of September 23

When you finish Tutorial 3, you should be able to

1. Write overloaded methods in a class
2. Override inherited methods from the `Object` class (or any class)

Not all problems in the tutorial have the same difficulty level. For each section of the tutorial, start with the problems that best match your level of Java expertise. It is recommended that you work on problems from each part of the tutorial. An *extra topics* section may be included for students that have already mastered all the material of a given tutorial.

You will be expected to use various Java classes in the tutorials and be expected to read their APIs and understand how to use them in your programs.

If you do not finish problems from each section of the tutorial in the lab, it is highly recommended to complete the problems at home (or in the lab) after your tutorial section. The more practice you have coding the easier coding will be for you.

For extra practice solving problems and writing Java code, solve the problems at
`http://codingbat.com/java`

0: HP 4155 [reminder]

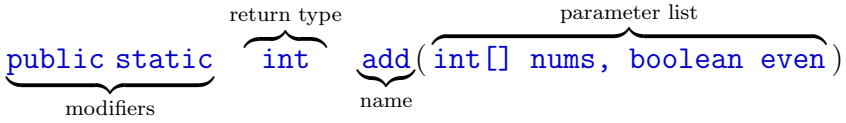
Here are some helpful reminders about using the Windows machines in HP4155.

`http://www.scs.carleton.ca/technical-support/windows`

In particular, be careful of where you save your data while working on the school machines. Lost data on the school machines because you did not move the data to your Z drive, dropbox, google drive, etc., before you logged off will not be accepted as an excuse for late or missed assignments. Please take care that you save your data somewhere that is external to the lab computer.

1: Overloading Methods

As you have seen in Problem 4 of Assignment 1 (Pythagoras), we can have methods in Java with the same name. This is called method **overloading**. In Java (and other languages), a method is uniquely specified by its **signature** and not just its name. So far in the course[1], a method declaration in Java looks like



The **signature** of the method consists of the method name and the parameter list. This uniquely defines a method. Therefore, we can have multiple methods with the same name as long as they have different parameter lists. This is called method overloading.

For example, a Java class might have the following methods defined:

- `public static void foo(int x);`
- `private static int foo(boolean b);`
- `void foo();`
- `public String foo(int x, String s);`
- `public String foo(String s, int x);`

All of these methods would be valid. Notice from the last two examples that the order of the types in the parameter list matters. (The names of the input parameters does not matter.) The following pairs of function declarations are not valid.

- ```
  void foo(int x);
  long foo(int x);
  ```
- ```
  public String foo(int x, int y);
  public String foo(int y, int x);
  ```

Each of these examples would result in a compiler error. For the first example, you would receive an error message similar to `foo(int) is already defined in T`.

➡ Write a Java class called `Tut2` with the following methods in it:

- `public static int age(int x)`
- `public static int age(String s)`
- `public static int age(byte[] b)`

Each of the methods takes some input that represents the age of a person in years and outputs the (approximate) age of the person in days. Assume that today is person's birthday and don't worry about leap years. The first method assumes that the input is the number of years, the second assumes that the number is inputted as a String (so 10 is inputted as "10", and not "ten")), and the last method assumes that the age in years is the sum of the bytes in the input array. Test your methods with your `main` method.

[1] We will see that there is more to a method later when we talk about exceptions.

2: The Object Class

The Object class in Java is the root class for all classes. Every object that you use in Java, except arrays, inherits from the Object class. This means that all of the public (and protected) non-static attributes and methods that are in the Object class are also in every object (other than arrays) that you use.

Look at the Object API and see that is in it.

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

➡ Write a Java class called Age with just a main method. Copy and compile the following:

```
public class Age{

  public int age = 3;

  public static void main(String[] args){
    Age ice = new Age();
    Scanner keyboard = new Scanner( System.in );

    System.out.println(ice);
    System.out.println(keyboard);
  }
}
```

Ok, correct the initial compile error and then run this code. What do you notice about the output? It is probably not that meaningful to you. In general, Java does not know how to print (display) your data. We have been misled so far, because it does know how to print primitive data types and Strings and that is pretty much all we have been using. Most Java programs will be using other things (objects) though. And how could Java possibly know how to print every object that will be used? (Especially when most object types (classes) were not even invented when the println method was written!)

Now add the following two lines of code to your class just after the print statements.

```
System.out.println(ice.toString());
System.out.println(keyboard.toString());
```

What do you notice when you run your program? What is going on here?

It turns out that the input parameter to println (and print) is a single String. When you print something, Java will convert your input into a String. If the input was a primitive data type, it just does the conversion automatically. If the input was an object, it calls that object's toString() method. Take notice that you didn't write a toString() method for the Age class though! Of course, you didn't have to. Since the Object class has a toString()

method, and since all objects inherit everything that Object has, your object also has a toString() method.

Unfortunately, the toString() method, as specified in the Object class does not print out anything that is very human-readable. It does contain information, but that information is probably not what you wanted to see when you printed the ice object.

This is where method **overriding** comes in handy. In Object Oriented Languages, like Java, you are allowed to redefine inherited methods. This is called method overriding.

➡Look in Object's API for the toString() method. In your Age class, write a method called toString(), with the same modifiers, return type and signature as that found in the Object class. In the body of your method, just put `return "age = " + age;`. Now compile and re-run your program.

What happened? When you print any Age object (which ice is), Java will automatically execute the toString() method that you just added to the class to convert the object to a String. As we create more classes (which is pretty much all that we'll be doing for the rest of the year!), you have a way of printing each class in any way that we want. This is handy.

➡ Modify your Age class to have another attribute that is an array of integers called. `data`. Modify your toString() method so that each time you print an Age object (like ice), it prints something like `Age is 12; data = {2,14,199,-2}`, if age=12 and data = {2,14,199,-2}.

➡ We have to be careful with arrays though. While we know that an array is an object, it is not an object that is instantiated from a class. Therefore, it does not inherit the methods from Object like most objects will. There are classes that will help you print an array. In general, just iterate over your array and print each item one at a time.

3: More of the `Object` Class

The `Object` class has two more methods that we will mention: `equals()` and `hashCode()`.

➡ Write a class called `Bunny`. The class should have two attributes: a String called name and an int called age. In the main method of your class, create 2 bunny objects and set their attributes to be the same. Using the `==` operator, test if they are the "same" bunnies or not.

➡ Override the `equals()` method so that two bunny objects are considered the same when their names are the same and they have the same age.

➡ Override the `hashCode()` method so that it returns the age of the bunny plus the length of the String that is the bunnies name. Now, print out the bunnies using println. What do you notice? (See the toString() in Object's API.)

➡ Override the `toString()` method in your Bunny class so that it prints something meaningful (such as the bunnis name and age).

4: Inheritance 101

When a class (the child class) inherits from another class (the parent class), all of the attributes and methods (except the constructors) that the parent class has are automatically also in the child class.

➡ Create a Java class called `Parent`. You class should have two attributes, name and x (a String and an int), and one method

```
public int add(int n){...}
```

which adds the input `n` to the integer attribute, x, of the object. Create another class, called `Test`, with a main method to test your class. Instantiate some Parent objects, and use the add() method. Something like

```
Parent p = new Parent();
Parent q = new Parent();
p.x = 2;
q.x = 5;
System.out.println(p.x + q.x);
System.out.println(p.add(10));
System.out.println(p.add(10));
```

➡ Override the `toString()` method so that when you print a `Parent` object, it says something meaningful (like the name).

➡ Create a Java class called `Child` that `extends` the `Parent` class from above. How do we extend a class? When we declare the class, we use the Java keyword `extends` and name the class that we are extending:

```
public class Child extends Parent{
  ...body of class...
}
```

Leave your Child class empty for now. Modify your `Test` class by first duplicating the entire body of the main method (so it appears twice) and then changing all instances of the word "Parent" with "Child" in the copied part. That is, create some Parent objects and some Child objects.

Run your program. Notice that there is no difference beteeen using a Child or Parent object. This is because we did not add anything to the Child class, it is essentially identical to the Parent class. Each have the same attributes and the same add method. Add lines of code to print out both Parent and Child objects using println. What do you notice?

Since Child extends Parent, it gets everything that was in Parent. Since Parent changed the toString() method, Child also got this modified version of toString().

➡ Add an attribute (integer y) to the Child class. Experiment in your Test class with this new attribute (access it, change its value, print it to standard output, etc). Try to access this y attribute in a Parent object. What happened?

Override the toString() method in the Child class so that it is different from the method in the Parent class. For example, it might be `return ‘‘I am a Child.  My name is’’ + name;`. Modify the Parent class' toString() to mention that it is a Parent in a similar way.

➡ What happens when we run this code?

```
public static void main(String[] args){
   Parent p = new Parent();
   p.name = "cat";
   p.x = 12;

   Child c = new Child();
   c.name = "dog";
   c.x = 3;
   c.y = 77;

   Parent pc = new Child();
   pc.name = "eel";
   pc.x = 15;

   System.out.println(p);
   System.out.println(c);
   System.out.println(pc);
}
```

What is happening here? We will be discussing inheritance and object behaviour more in class soon.

5: More Coding

If you already knew the material of this week's tutorial or has finished it very quickly, here are some problems to practice your coding skills. The first two problems are modifications of the extra coding problems from the last tutorial. he last problem is from one of the author's of the Sedgewick & Wayne textbook. This is a fun little project if you are skills are advanced at this point.

➡ Write a class with a method called `primes`. The method has a `long` input parameter, call it n, and outputs an array of `long`s. The method computes all prime factors of the input

number and returns them in the output array. Recall that a prime number that is evenly divisible by only 1 and itself. Assume that $n \geq 1$.

For example, `prime(2*17*101)` should output the array `[2,17,101]`.

➡ Write a class called `Lagrange4` which uses one command line argument (a positive integer, which we'll call n). Your class will find and display four non-negative integers a, b, c, d such that

$$n = a^2 + b^2 + c^2 + d^2$$

When the four numbers are found, your class will display these numbers to standard out. For example, running `java Lagrange4 41` will display `6 2 1 0`.

How much work is your program doing to solve this? How can you reduce this work? Try to minimize the number of iterations each loop does in your algorithm. How many loops are you using?

☞ See `http://en.wikipedia.org/wiki/Lagrange's_four-square_theorem` for more information about this.

➡ Work on the Guitar Heroine problem.
`http://nifty.stanford.edu/2012/wayne-guitar-heroine/`