Computer Science 202
Introduction to Programming
The College of Saint Rose
Fall 2013

Topic Notes: Java Applications

While we have been and will continue to keep a graphics theme to our course this semester, not all programs are graphical or even event-driven. We will take a short break from all of that to consider some programs that operate on textual and numeric data, read from the keyboard and printed to the screen. Later, we will see how we can also interact with files on our computer's hard disks or network devices.

Java Applications

The programs we have considered so far, those which have a class the "`extends WindowController`" are called Java *applets*. These usually have a graphical user interface (ours has been a simple one so far), and are designed to be able to be embedded into web pages like is done for class examples. The other class of Java programs are called or *console applications* or sometimes just *applications*.

As you are already familiar with some of Java's syntax, we will proceed by looking at a few examples and discussing what is different from the kinds of programs we've seen so far.

Let's start by looking at a "Hello, World!" application program in Java. It is a longstanding tradition among programmers to write their first program in any language to print out that message, and this is different enough from what you've been doing that we'll start there.

```
/*
  A Hello World example in Java
*/
// class example
public class HelloWorld {

   public static void main(String[] args) {

     System.out.println("Hello, World!");
   }
}
```

It starts out with some comments, just as you've been (hopefully) doing.

Here, our class does not `extend` anything. In particular, since it doesn't extend `WindowController`, it will not have a graphical window or access to a canvas. It also will not know how to respond to mouse events.

Instead of a `begin` method, the first method that will be invoked is called `main`:

```
   public static void main(String[] args) {
```

This particular method definition starts with 3 keywords: `public`, `static`, and `void`. The meanings of each of these will become more clear with further examples.

It then has the name of the method, `main`. This is an identifier. `main` has a special meaning here because that is the name of the method that will be executed when we run our program. In general, methods can use any name that is not a keyword. However, we will see that there are *naming conventions* that give some rules about how methods and other identifiers should be named.

All method definitions have a list of *parameters* inside parentheses. We saw parameters to our mouse event methods, and you will be seeing much more about methods and parameters soon. For now, we can just follow the rule that our `main` method must have a parameter of `String[] args` as this one does.

In our case, the method body consists of just one statement:

```
  System.out.println("Hello, World!");
```

To type in and run this program is similar to what we've been doing all semester, but different enough that it's worth going through.

1. Launch BlueJ, create a project as we have been doing all along.
2. Click "New Class..." and enter a class name, but instead of "WindowController" class, you will choose "Application Class".
3. Click on the class icon that appears, and you should see the skeleton of a Java application rather than an "WindowController".
4. Add the printout from above to the provided `main` method.
5. Compile as you have been doing by hitting the "Compile" button.
6. Back in the project window, you can right-click on the class icon to bring up a menu. It won't have the "Run Controller' option. Instead, choose "`void main(String args[])`". An extra dialog window will come up. For now, you can just hit "Ok".
7. You should get your message in the Terminal window.

Glorified "Hello, World!"

The simplest class of programs are also the most excruciatingly boring – ones that just print out the same message or set of messages every time we run them. Such programs are rarely useful in practice and really serve only to introduce the basics of a programming language.

But let's look at one anyway.

See Example: Seuss

This is basically `HelloWorld` all over, but there are a few little items that are new.

- Notice that the sentence "We know how." is printed on the same line as "Well, we can do it.". That's because we used a different printing method for the latter: `System.out.print`. This one works the same as `System.out.println` except that the output is not advanced to the next line at the end.
- The last statement includes some *escape sequences* that cause the output to be formatted a bit differently than it would otherwise appear. Escape sequences begin with a `\` character and are followed by a *control character* that defines the behavior of the sequence. Here, we have three:
    1. `\t` inserts a "tab" character, effectively indenting our output in this case,
    2. `\n` advances the output to a new line, and
    3. `\"` prints the double quote character, which would otherwise be impossible since a regular `"` character would be interpreted as the end of the text we are trying to print.

One bit of terminology at this point: the methods `System.out.println` and `System.out.print` are part of the *Java API* (Application Programmer Interface). Any valid Java installation comes equipped with an extensive collection of pre-written software that our programs can use.

These are standard on all Java installations. We have also been making use of the ObjectDraw library, which requires some extra setup.

Interactive Programs

To create nearly any interesting program, we need to be able to provide it with *input*. This will allow the program to react differently when presented with different inputs.

See Example: HelloYou

It might not seem like this should be much more complicated than our previous programs, but it turns out that to do this in Java, we need to utilize a number of new ideas and Java constructs.

First, we need to figure out how to get information from the keyboard into our program. To do this, we again turn to the Java API. There are several mechanisms available, a few of which we will see this semester. But we will start with one called a `Scanner`.

In order to use a `Scanner`, we will need to tell Java that we intend to use it, by inserting the line:

```
import java.util.Scanner;
```

at the top of our program (before the class header). We will see later how to determine exactly what to "import" to use various Java API functionality, but for now, just know that this is what we need to do to use a `Scanner`.

Then, in our `main` method, where we wish to access information from the keyboard, we *construct* a `Scanner` that we can use in our program. This is done with the line:

```
Scanner input = new Scanner(System.in);
```

There's actually quite a bit going on in this line, and we'll examine it more carefully in a minute. But for now, we're creating a `Scanner` that uses `System.in` (which is Java's cryptic way of saying "what is typed at the keyboard"), and giving it a name, `input`. `input` is a *local variable* – one which exists only within `main`, unlike the *instance variables* we had in our graphical classes that exist in all methods of our class.

Now that we have a `Scanner` called `input`, we can ask it to give us the next chunk of text that was typed at the keyboard. There are many possibilities for what we mean by "chunk" but for now, we just want the word that someone types in as their name. Java's `Scanner` provides a method that does just that, called `next`.

We will also need a name for the word that was typed in, so we can print it out later. This is all accomplished with the line:

```
String name = input.next();
```

Before we carry on further, we need to consider the concept of variables a little more closely.

We know that a *variable* is a named storage location in the computer's memory. We use a variable when we have determined that there is some piece of data that we have in one Java statement, and we need to remember that information for use in later statements.

When we need a variable in our program, we must *declare* that variable to Java, which is just a fancy way of saying that we are going to introduce a name to our Java program and tell Java what *type*, or kind, of data we intend to store there. We have been doing this with instance variables already, where we place the declaration at the top of the class, outside of any method, and include the directive `private`.

A local variable declaration takes one of two forms:

```
type name;
```

or

```
type name = initialValue;
```

where "type" is the *data type* (or " kind of data") we will store, and "name" is the identifier we intend to use to refer to that data. In the second form, we also *initialize* the variable to have a specific value.

We will see many data types that store a variety of kinds of information. For now, we have two:

- `Scanner` – which is the keyboard input mechanism we wish to use
- `String` – a collection of text, like a word or sentence

We give our `Scanner` the name `input` and the `String` the name `name`.

When naming our local variables, we need to keep in mind the same considerations we used for instance variables:

- The name must be a valid Java identifier. This means it must consist only of letters, numbers, the dollar sign character, and the underscore character (though it can only start with a letter).
- The name should follow Java's naming conventions. Recall that for variables, we use lowercase letters, except when we have a name that is made up of multiple words, in which case we capitalize all but the first word.
- The name should be meaningful. That is, it should give some indication of what the variable is to be used for. The names here satisfy that requirement: `input` implies that this is where we get our input, and `name` implies that this is the name of something.

Once we have a variable, we can make use of its value later in our program. We do that here when we call the `next` method of our `Scanner` named `input` and when we use the `String` named `name` in the `System.out.println` statement at the end of our program.

Another Example, Adding in `JOptionPane` I/O

We will next consider another Java application, but with the added bonus of using a different mechanism for input and output.

The problem: we wish to write a program that calculates the number of full payments needed for a no-interest loan where we are given a loan amount and desired monthly payment. This number is reported. If additional funds are due after those full payments are made, that is reported as well.

See Example: NoInterestLoan

The comments in that example describe in detail three new items:

- The use of `JOptionPane.showInputDialog` to bring up a dialog box with a message and a text box for input, and returning the text typed into the box as a `String`.

- The use of Java's `Integer.parseInt` method to convert a `String` to an `int`, which is necessitated here because the `JOptionPane.showInputDialog` only returns `String` values.
- The use of Java's `JOptionPane.showMessageDialog` to bring up a dialog box to display some program output.

Use with Applets

There is nothing special about Java applications in their ability to use `Scanner`s and `System.out.println` or the `JOptionPane`s for I/O. We can do the same with an Objectdraw program (in this case, just using a single call to a `JOptionPane,showInputDialog`):

See Example: BasketballName