Computer Science 202
Introduction to Programming
The College of Saint Rose
Fall 2012

Topic Notes: Beginning with Programming

We will next take our first shot at writing a programs to solve specific problems.

Analyzing a Problem

Let's consider the following situation and think about how we might develop a program to solve it:

Suppose you are a teacher with a class of 20 students. You will give some number of tests (3, 4, or 5) during a marking period. Write a computer program to print each students name and marking period average.

We first need to *define the problem* carefully.

- What problem are you trying to solve?
- Understand expected inputs
- Understand expected outputs

We will consider two approaches to developing a plan to solve the problem.

Pseudocode Planning

We can describe the problem in English-like phrases that we will call *pseudocode*.

1. Begin with first student.
2. Set number of tests counter to zero.
3. Set student test total to zero.
4. Get student's name.
5. Get a test grade.
6. Add 1 to the number of tests counter.
7. Add the test grade to student test total.
8. Repeat steps 5-7 for the number of tests in marking period.

9. Calculate student's marking period average.
10. Print student's name and marking period average.
11. Next student.
12. Repeat steps 2-11, until you have done all students in class.

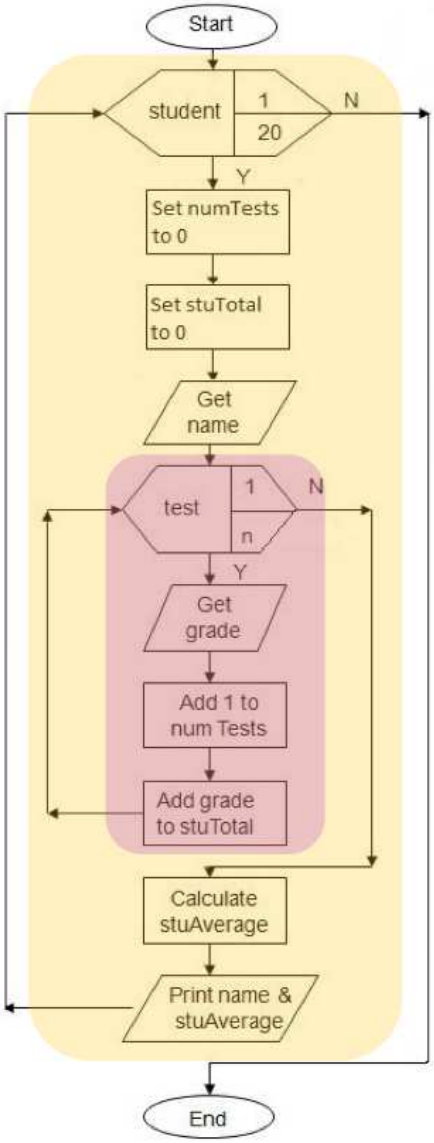
This is like a set of instructions or a recipe for solving the problem.

Flowchart Planning

Another, more visual approach to planning is called a *flowchart*.

Here, we show the steps in the algorithm as a diagram (using specific shapes to add to meaning). A flowchart shows the sequence of statements or *flow of execution*.

See supplied appendix for an explanation of symbols. [pages 866-879 from Computer Science, A Structured Approach Using C++, 2nd Edition, by Forouzan and Gilberg, Brooks/Cole, 2004.] ** available in Blackboard only.



- This diagram clearly shows the flow of execution.
- One group of statements is repeated for each of the 20 students.
- A smaller group of statements is repeated for each test.

Either pseudocode or a flowchart can help you plan a solution. You are encouraged (and sometimes required) to use a flowchart in this class.

Turning it into Java

Once we have a plan in the form of pseudocode or a flowchart, it's time to write a program to perform the task.

We mentioned earlier that there are many programming languages out there and that we will be using Java.

- We could use a *low-level* or *machine language*
  - machine code can be used on just one type of computer (CPU)
  - provides the fastest execution
  - consist entirely of binary numbers representing opcodes and addresses
  - very difficult, error prone, and rarely done
  - not really intended to be done by human programmers
- An *assembly language* is close to machine language but is designed to be used by a human
  - uses some word-like symbols called *mnemonics* for opcodes
  - can use both decimal numbers and labels for memory addresses
  - assembly language programs are translated (assembled) into machine language
  - rarely used outside of operating systems, embedded systems
- A *high-level programming language* is a better choice
  - *procedural* languages such as FORTRAN, COBOL, BASIC, Pascal, C, and C++ dominated until the mid 1990's
  - *object-oriented* languages such as C++, Visual Basic, Java, and C# were developed as graphical user interfaces (GUIs) became more common

But this course is about programming in Java, so that is what we use. Since we don't know any specifics of Java yet, all we will do is look at a solution and run it to see an example.

See Example: StudentAverages

Java Basics

We will look at a series of simple Java examples to familiarize you with the basics of the Java programming language, including how to develop and run programs.

Let's start by looking at a "Hello, World!" program in Java. It is a longstanding tradition among programmers to write their first program in any language to print out that message.

```
/*
  A Hello World example in Java
*/
// class example
```

```
public class HelloWorld {

   public static void main(String[] args) {

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

If we were to think about this very simple problem in English or psuedocode, it would be something like:

```
print the message "Hello, World!"
```

Like many programming languages, even the simplest Java programs require a good amount of additional code beyond the line that performs the task.

That is because a programming language is very picky in how a program is formatted. This is part of the *syntax* of Java. Let's look line by line.

The first line is a *comment*. This is not a part of the program that actually gets executed – it is just some text that is there for the benefit of anyone reading or modifying the program. You will be required to make extensive use of comments in your programs this semester, even though they have no effect whatsover on the correct execution of the program.

In Java, comments can take one of two forms.

- Any text between `/*` and `*/` is a comment, possibly spanning multiple lines.
- Any text on a single line following `//` is a comment.

In Java, programs are written by defining *classes*. We will see more about what it means to be a class later in the semester. For now, we just need to be aware that the next line tells Java the name of the program.

The words `public` and `class` are special words called *keywords*. These are words that have special meaning to Java and cannot be used for other purposes.

All keywords in Java must be typed in lower case because Java is a *case sensitive* language. That is, `public` is a keyword, but Java would not recognize it as such if you typed `Public` or `PUBLIC`.

Page 10 of the text has a list of Java keywords.

We said the word `HelloWorld` is the name of the program. It is an example of an *identifier*. This is a programmer-defined name used for a variety of purposes. This one is used only in one place.

Punctuation is also an essential part of the syntax of Java. The end of the line defining the class name has a `{` character. This along, with the corresponding `}` character at the end of the program defines for Java the extent of the class definition for the class named `HelloWorld`.

Inside that class definition, we have a *method* definition. This starts with

```
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 will say much more about 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.

Inside the `{` and `}` brackets is the *body* of the method, consisting of one or more Java *statements*. A statement is a complete Java instruction that causes the computer to perform an action.

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

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

There are many kinds of Java statements. This one is a call to one of Java's builtin methods. The line says to call the method `System.out.println`. A method call includes the name of the method and then the parameters to send to that method inside parentheses. This method expects one parameter: the text that it should print out. In this case, we just place the text we want to print inside of double quotes. We will see more intersting examples soon.

Our Java statement ends with a semicolon. This is an important part of Java syntax that tells the language that one statement is done.

An important part of learning to program in Java is to learn when and where to use the punctuation.

It is important to realize that there is a difference between a *line* of Java code and a Java *statement*. There is nothing wrong with having a Java statement that spans multiple lines. We could have just as well typed:

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

and Java would do exactly the same thing. Hopefully this will give some indication of why the semicolon is so important. Just ending a line is not enough to tell Java that the statement is complete.

Let's augment our example to show one more important Java construct: the *variable*.

```
/* A Hello World example in Java */
```

```
public class HelloWorld {

   public static void main(String[] args) {

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

The only change we've made is to replace the printout with 2 lines. The first defines a variable:

```
   String message = "Hello, World!";
```

This Java statement defines a name `message` which should contain data of type `String`, and have it contain the text `"Hello, World!"`.

What does all that mean? Well, `message` is an identifier. It's a name we are giving that will let us refer to some piece of data by that name later in the program. You can see that the name is also used as the parameter to the `System.out.println` statement on the next line.

`String` is a builtin data type that can hold some text. If instead we wanted to store an integer, we could define a variable like this:

```
  int count = 40;
```

Here, `count` is the name for a variable that can hold an `int` (Java's data type for an integer value), and gives it the value 40.

Defining a variable basically instructs Java to reserve some part of the computer's memory for use by our program, and defines a name we can use to refer to that part of memory.

The computer's memory at a low level is really just a huge collection of locations that can hold numbers, and these locations are referred to by number. One of the great advantages of a high level language like Java is that we can refer to memory using variable names, and that the language will take care of mapping names to locations and making sure the same location is not used for multiple purposes.

If we think of the computer's memory as being analogous to a large wall of numbered mailboxes, defining a variable is like reserving one of those mailboxes for a person, and adding that person's name to a list so that that person can tell which mailbox is his, and incoming mail addressed by name can be routed to the correct box number.

Lab Activity: Writing and Running a Program in BlueJ

The Programming Process

Now we have seen all parts of the programming process we will be using all semester.

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools (flowchart/pseudocode) to create a model of the program.
4. Check the model for logical errors. Do you get a reasonable answer?
5. Type the code and compile it.
6. Correct any errors found during compilation.
7. Repeat previous 2 steps as many times as necessary.
8. Run the program with test data for input.
9. Correct any errors found while running the program.
10. Repeat previous 5 steps as many times as necessary.
11. Validate the results of the program. Does the output make sense?