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

Topic Notes: Java Fundamentals

We will proceed using a series of increasingly-complex problems to practice our problem solving skills and to introduce and reinforce Java constructs and terminology.

By the time we get through this series of examples, you will have seen nearly everything presented in Chapter 2 of Gaddis, though we will do so in a different (hopefully, better) order.

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.

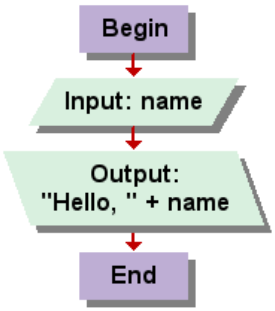
One bit of terminology at this point: the methods `System.out.println` and `System.out.print` are part of the *Java API* (Application Programmer Interface). Any valid Java installation comes equipped with an extensive collection of pre-written software that our programs can use.

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 outputs.

Before we see how to do this in Java, we will make our first real use of Visual Logic.

We will develop the Visual Logic flowchart and run it.



Once we are convinced that our logic is correct (and that's not hard, with such a simple problem), we can go ahead and develop a Java program.

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 *variable*,

which is a fundamental construct in nearly any programming language, and again one that we will examine more carefully in a moment.

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.

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.

A 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 variables, we need to keep in mind several considerations:

- 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.

One last new idea here is that we now have something more complex as the text to be printed by `System.out.println`. It's not just some text in double quotes, but some text in double quotes, followed by a `+`, followed by the name of our `String` variable.

This is an example of *string concatenation*. We have the *string literal* (*i.e.*, some text inside double quotes) to which we "append" the text in the variable `name`.

Working With Numbers

Computers often do just that: they compute with numbers. So next, we consider some examples of programs that work with numbers.

Integer Values

We start simple. Let's compute a rectangle's area and perimeter.

We first build a flowchart to make sure we know how we will approach the problem.

Then we can covert this to Java:

See Example: Rectangle

There are a few things to note in this program.

First, we are working with numbers rather than words. This changes how we read the data from the keyboard through our `Scanner` and the type of variable we need to declare to store that data.

For this example, we are requiring that the dimensions of the rectangle are integer values.

The Java type we will use to store an integer value is called an `int`. We declare and initialize `int` variables named `width` and `height` to store the rectangle's dimensions.

`int` is one of Java's *primitive data types*. We will see several other examples. These are the only types that are usually specified with an all-lowercase keyword.

We next need to use a different method of `Scanner` to force it to look for an integer and return it in as a Java `int` instead of a `String`. That method is called `nextInt`.

Once we have our width and height, we need to compute the area and perimeter from them. For

this, we need to declare two more `int` variables and perform some computation to compute their values.

If you remember your elementary school geometry, you know that to compute the area of a rectangle, we multiply its width by its height. And to compute the perimeter we add up the lengths of all sides, which in this case is twice the width plus twice the height.

Java uses a notation to specify mathematical computations (a mathematical *expression*) that is mostly familiar from math. As we can see from the statement that computes `area`, we use the `*` operator to specify multiplication.

So that statement instructs Java to multiply together the `int` value found in the variable `width` by the `int` value found in the variable `height` and store the product in the `int` variable `area`.

The computation of `perimeter` is a bit more complicated, but still pretty straightforward. We see that addition is specified by `+` and that we can use numbers in our expressions as well as values stored in variables.

We do need to know in what order Java will perform the operations here. If it does `2 * width`, then adds 2 to that result, multiplying that resuly by `height`, we will get the wrong answer. Fortunately, Java follows a strict *order of operations*. In this case, we say that multiplication has a higher *precedence* than addition, so Java will compute `2 * width`, then `2 * height`, then add together those results.

We will look in more detail at order of operations as we encounter other mathematical operators in subsequent examples.

Finally, we print out our results. We can see here that Java "does the right thing" when we concatenate string literals with `int` values.

Question: what happens if we type in something that's not a valid `int`?

Floating-point Values

Our next example, which will be developed in class, is to perform a simple miles per gallon computation. Again, we will prompt for inputs, compute our answer, and report the result.

What do we need to know to make this example work?

- If we want to store non-integer values, which are called *floating-point values* in Java, we use variables of type `double` instead of `int`.
- If we want to read in `double` value from a `Scanner`, we use the `nextDouble` method.
- Division is specified by the `/` operator.

See Example: MilesPerGallon

Note the difference between integer division and floating-point division by trying the above first with `int` data, then with `double` data.

When we divide two `int` values using `/`, the result is the *quotient*, and we throw away the remainder. If we want the remainder (and only the remainder), we can use the `%` operator, often called the "mod" operator as it performs modulo arithmetic.

Any division operator where both operands are `int` values, results in an `int` quotient. If *either* operand (or both) is already a `double`, the results is a `double` and the answer would include any fractional part as a decimal.

Operator Precedence

We can specify complex arithmetic expressions using any combination of the following:

`*`	multiplication
`/`	division
`%`	remainder
`+`	addition
`-`	subtraction

In a long expression such as

```
12 + 9 / 4 - 18 % 4 * 19
```

there are choices to be made in how to evalulate. Fortunately, Java makes these decisions and makes it clear to us how it will evaluate such an expression.

1. *unary* negation operators are applied first, working left to right if there are multiple such operations
2. multiplications, divisions, and remainders are computed, again left to right
3. additions and subtractions are computed, left to right

So in the above expression, we first check for unary negations, and there are none.

Then, we do the multiplication, division, and remainder operations. Since these are all integer values, the any division will be computed as an integral quotient.

So, the `9 / 4` evaulates to `2` first. Giving

```
12 + 2 - 18 % 4 * 19
```

Next, `18 % 4` is evaluated to `2` (the remainder when we divide 18 by 4). Giving:

```
12 + 2 - 2 * 19
```

One multiplication remains, so we compute the `2 * 19` as `38`, giving:

```
12 + 2 - 38
```

We are left with only additions and subtractions, which are evaulated left to right. `12 + 2` becomes `14`, leaving us:

```
14 - 38
```

and after the last subtraction, we have `-24` for a final result.

The same rules apply if we have data in variables declared as either `int` or `double` values.

If we wish to override the default rules, just like in math, we can place parentheses around any lower-precedence operation that we wish to have performed before some higher-precedence operation, or if we want to change the order among same-precedence operations to do some further to the right before some further to the left.

Named Constants

The next program, which we will develop in class is going to do the following:

- Read in 2 lines of input. Each contains the name of a baseball team (which must be a single word) and the number of runs that team scored.
- Report the total runs scored.
- Report the average number of runs per inning, both as a decimal and as a mixed number (a whole number followed by a fraction).

See Example: RunsScored

This example is the first one that demonstrates an important feature of good programming style: the use of *named constants*.

In this case, we are going to assume a baseball game has 9 innings. But perhaps in some other cases, there are 6 or 7 inning games. So we define a constant:

```
final int NUMBER_OF_INNINGS = 9;
```

As our programs become more complex, we will be using many numeric values. Using many somewhat arbitrary numeric values in a program can make the program difficult to understand and modify. We can improve the situation by associating the values with names so that we are reminded what the values signify when we see the names used.

Java includes a mechanism to enable us to use such names effectively. If you include the word "`final`" in a variable's declaration, this indicates that the value assigned to it in the declaration will never change. This means that its value cannot be changed (possibly by mistake).

It also means that we can change the value in just that one place if we decide to change the number of innings in a game (at which point we would have to recompile our program). Otherwise, we'd need to remember to change all instances of the number if we changed any.

A couple other notes from this program:

- We read both a `String` and an `int` from the same keyboard input line.
- We need to be careful that the addition of the two scores are done before the string concatenation in our printout.
- We need to be careful that the result of our `int` division does not throw away any fractional part in the case where we want to store the result in a `double`.
- We use both integer division and the remainder operator to compute our mixed number result.