Computer Science 523
Advanced Programming
The College of Saint Rose
Summer 2014

Topic Notes: Methods

As programmers, we often find ourselves writing the same or similar code over and over. We learned that we can place code inside of loop constructs to have the same code executed multiple times. There are other situations where we wish to execute some of the same code repeatedly but not all in the same place in our program's execution.

Consider a simple program, which we could have written months ago, that prints the lyrics to *Baa Baa Black Sheep*.

See Example: BaaBaaBad

Notice that we have 4 printouts repeated – these are the refrain of the song. To accomplish this, we either need some copying and pasting or we need to re-type some lines.

A loop wouldn't help us here, as the lines we need to repeat are separated by 4 other lines that do not repeat.

Fortunately, all modern programming languages, including Java, allow us to group sets of statements together into units that can be executed "on demand" by inserting other statements. These constructs go by many names: *functions*, *methods*, *procedures*, *subroutines* or *subprograms*.

Java calls them methods.

Here is our example, using a Java method to group our repeated statements.

See Example: BaaBaaBetter

Our method for printing the refrain looks a lot like the method we know well: `main`. Other than the name (which in this case is "`refrain`") and the fact that we don't have the "`String args[]`" in the parentheses, it has a very similar *method header* to the `main` methods we have been writing all semester.

Just like `main`, the *method body* of the `refrain` method consists of a collection of Java statements that will execute when the `refrain` method is called.

We can also see in the `main` method where we *call* the `refrain` method. We simply put its name, followed by `();`

This is somewhat similar to what we have been doing in earlier examples to call methods:

```
System.out.println("Hi!");
keyboard.nextInt();
JOptionPane.showMessageDialog(null,"Hi!");
```

except that there is no name or names before a period before the method name. We can omit it here, because that means we want to call a method in the same class as the method which is making the call.

We could have made our method calls to `refrain` look like the ones we've been using all along:

```
BaaBaaBetter.refrain();
```

Note that we used a method in this case primarily because it allowed us to reduce the amount of repeated code. This is in itself a worthy goal. If we had misspelled one of the words in the refrain, we can change it in one place and it will be corrected in both printings of the refrain.

However, there is another advantage in readability. By having 2 calls to the `refrain` method in our `main` method, it is more clear what we are doing there. With this in mind, we can consider moving parts of our `main` method into a separate method just for clarity.

See Example: BaaBaa2Methods

Passing Parameters to Methods

Some methods work like the ones in the previous examples: they simply perform the same exact task every time they are called.

However, many others will perform functionality that depends on some input. The way we get input to a method is by passing *parameters* (also known as *arguments*) to a method.

We have done this with the methods we have been using all along. What does `System.out.println` print? Whatever we pass as its parameter. What is the range of values returned by a `Random`'s `nextInt` method? It depends on the parameter we pass.

We can pass parameters to methods we write as well.

See Example: NumberInfo

We essentially introduce a variable (which is called a *formal parameter*) to our method that is initialized to whatever value is passed in the parentheses when we call the method (which is called an *actual parameter*).

A slightly more complex example:

See Example: HoursWorked

Here, we pass a `String` parameter to our method. It contains the contents of one line of an input file, and the method is responsible for breaking down that line into its components, which are an employee id number, an employee name, followed by some number of floating point numbers that represent hours worked by day.

This also demonstrates another use of a `Scanner`. If we pass a `String` as the parameter to a `Scanner`'s constructor (recall that we normally pass either `System.in` for keyboard input, or a `File` to read from a file), we can use the `Scanner` methods to read individual words or numbers.

We could alternately move the reading of each line of the file into our method as well.

See Example: HoursWorked2

Now, the parameter to our method needs to be the `Scanner`, so the method will be able to call the `Scanner`'s `nextLine` method.

Passing Multiple Parameters

Nothing stops us from passing multiple parameters to a method or passing information of any data type.

See Example: SumOfSquares

Here, we create a method that accepts two parameters.

Order matters when passing parameters. The first parameter in the call will "match" the first parameter in the method signature, the second with the second, and so on. In this example, we'd get the same result, but that is not the case, in general. Order would matter, for a simple example, if we changed to a "difference of squares" here.

Returning Information from Methods

So far, each method we have written has the same start to its signature:

```
public static void
```

We will now change that last word to go from a "void" method – one which does not return any information, to one which does.

We have used methods that return values all semester, but we have not written any ourselves. Consider some of the following methods:

- `Integer.parseInt`
- `JOptionPane.showInputDialog`
- `nextDouble` of a `Scanner` object
- `nextInt` of a `Random` object

Each of these results in Java performing some task, and giving back to the caller some information.

We can write these kinds of methods as well.

Our first example will be a method that adds up all of the integers between 1 and a given number, and returns the sum.

So a call such as

```
int sum = sumNumbersTo(10);
```

should leave a value of 55 in `sum`.

Such a method and some examples of how to call:

See Example: Sum1ToN

A few quick notes about this example:

- Since our method computes an integer value, we replace `void` in its method signature with `int`. This is the method's *return type*.
- The value we compute that we wish to have our method send back to its caller is specified in a `return` statement. This is the method's *return value*.
- Any code in the method after a `return` statement will not be executed, so is not allowed. An exception might be if we have a `return` inside of a conditional statement (like an `if` or `switch`).

We can see immediately that we have some similar advantages to our `void` methods. The `main` method becomes shorter, and we avoid potentially having to repeat sections of code when we want to compute such a sum in multiple places in our program.

In this case, there is an additional advantage. Some of you may remember that there is a much easier (computationally speaking) algorithm for computing this sum. Rather than looping through all of the numbers and adding each to a running total, we could apply this formula:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

This is a more efficient operation, at least for larger numbers. Here, we do one addition, one multiplication, and one division. (Moreover, the division is a division by 2 - something computers are very good at.)

So if we discover this formula and want to change our program to use it, we need only change our method. We don't need to change anything in `main`!

See Example: Sum1ToNBetter

We next consider an example with a method that takes 4 parameters and returns a `double` value – one to compute the distance between 2 points in the plane.

See Example: Distance

The method itself is not that complicated, but the main program uses it several times, so the example looks more complex than it really is.

See the comments in the code for more.

A Utility Method for Input

We now will revisit a topic from earlier, using methods to provide a better solution. Recall that many of our programs that ask for input have had sections of code that look similar to this:

```
int val;
do {
   System.out.print("Enter a value between 1 and 10: ");
   val = keyboard.nextInt();
   if ((val < 1) || (val > 10)) {
     System.out.println("Value out of range, please try again.");
   }
while ((val < 1) || (val > 10));
```

We can write a method that can accomplish this, and make it generic enough to be useful in a variety of situations.

We will do this by modifying a program that computes a weighted average from a grading breakdown:

See Example: GradingBreakdown

This program works, but as you can see, the `main` method is quite lengthy and includes a significant amount of repeated code. Let's focus on that part of the code that prompts for an reads in category grades and does error checking on those inputs:

```
double labPointsEarned = 0.0;
do {
    System.out.print("How many lab points did you earn (total avail
        LAB_POINTS + ")? ");
    labPointsEarned = keyboard.nextDouble();
    if ((labPointsEarned < 0.0) || (labPointsEarned > LAB_POINTS)) 
        System.out.println("Response must be in the range 0.0 to " 
    }
} while ((labPointsEarned < 0.0) || (labPointsEarned > LAB_POINTS))
```

There are three items here that differ from one instance of this code block to the next:

- the name of the variable in which to place the result (`labPointsEarned` for this instance)
- the description of the category to be included in the prompt (`"lab"` in this case)
- the upper limit on the range of legal inputs (`LAB_POINTS` in this case)

If we are going to encapsulate this code block in a method, we will need to transfer these bits of information back and forth between the `main` method and the new method.

The description and the upper limit are both known to `main` and will be needed by the new method, so these will become parameters.

The “points earned” we are reading in will be read from the keyboard by the new method, but will be needed back in `main` to accumulate the overall average. This will become a return value.

Finally, our new method will need to know about the keyboard `Scanner` that `main` will still create. So the `Scanner` should also be passed as a parameter.

This gives us the result:

See Example: GradingBreakdownBetter