CS125 : Introduction to Computer Science

Lecture Notes #12
Method Syntax

Lecture 12 : Method Syntax

Methods

A *method* (also called a "procedure" or "function" or "subroutine" in some other programming languages) is a programming tool which allows us to label a set of instructions and reuse that set of instructions over and over by simply using the label instead of having to retype the instructions over and over.

We have already seen the use of a few methods:

- The first method we saw was the method named `main`, which is part of the "program skeleton" we've been using since lecture #3. In this case, the set of instructions that form the method are whatever code we've written inside `main()`.
- Next, we made use of `System.out.println();` and, after that, `System.out print()`. In both of these cases, we didn't write the set of instructions for printing; instead, we simply used the "label" – the actual `System.out.println();` or `System.out print()` line of code – and the instructions for printing were written into our program by the compiler.
- Finally, we made use of the `Keyboard.readInt()` method. In this case, the "label" was all you used in your own program, but you were also able to look inside the `Keyboard.java` file to see the instructions that that "label" told the compiler to run.

The method signature and method call

The code for a method (for example, what you find in `Keyboard.java`) is called the *method definition*, since it defines what the method does. The first line of the method definition is referred to as the *method signature*. The method signature is composed of three parts:

1. a *name* – just like variables have names, every method has a name as well. For example, `readInt` and `readDouble` are the names of two of the methods in the `Keyboard.java` file.
2. a *return type* – the type of value being returned. For example, `Keyboard.readInt()` sends back an `int` to whoever called the method. `System.out.println()` sends back nothing. When writing a method, you need to indicate the type of the value that is returned by the method (you can return only one value). If you don't return any value, then you indicate this via the "placeholder" type `void`.
3. zero or more *parameters* – a kind of variable specific to methods. These parameters appear within the parenthesis of the method, and when you use a method "label", you send in values to be stored in these parameters. You can then use the parameters as variables in the method definition.

This "label" that we use to activate a method is known as a *method call* or a *method invocation*, and it is said that we are *calling the method* or *invoking the method*.

So, we will give you a quick glimpse of an example of both a method signature and a method call, and then talk about the pieces in detail. This would be an example of a method signature:

```
int Add3(int x, int y, int z)
```

Note the name, `Add3`, the return type, `int`, and the parameters (which we'll discuss in just a moment) inside the parenthesis. One possible method call could be a code snippet like this:

```
int d;
d = Add3(3, 9, 1);
    ^^^^^^^^^^^^
```

The underlined code is the actual method call – values are placed inside the parenthesis to be stored in the parameters, and the method returns a value of type `int` (just like the `Keyboard.readInt()` method returns a value of type `int`) to be stored in the `int` variable `d`.

Our example in this packet will involve this method `Add3`, which does something simple (adds three numbers) so that we can concentrate on learning the actual method syntax instead of understanding a complex calculation.

Parameters

Listing a single parameter is very similar to declaring a variable, in that you need both a type and a name. We need the name so that we can refer to this parameter inside the method definition in the same way we would refer to any other variable. And, of course, all variables need a type, so that is why the parameter needs a type.

So, the form for listing a single parameter is:

```
paramType paramName
```

and the form for a list of parameters is to list single parameters, separated by commas. For example, if you had three parameters, those three parameters would be listed as follows:

```
type1 name1, type2 name2, type3 name3
```

For example, if we wanted a boolean, a char, and an int as parameters, we would pick names for these parameters and list them as follows:

```
boolean done, char oneChar, int value
```

Completing the method signature

The form for the method signature – which appears on the first line of a method definition – is:

```
return-type method-name(t1 p1,...,tn pn)
```

where

```
(t1 p1,...,tn pn)

      is

(type1 param1,...,typeN paramN)
```

We want our "`Add3`" method to accept three integers and return their sum. The sum of three integers will certainly be an integer as well. So, the first line of our `Add3` method should be as follows:

```
int Add3(int x, int y, int z)
```

In this method, the return type is `int`, and the names of our three parameters of type `int` are `x`, `y`, and `z`. We will return to this method in just a bit to finish writing its definition.

Calling a method

The information stored in the method signature is the only information you need to know in order to *use* the method. We can't call the method unless we know its name, knowing the return type lets us know how to make use of whatever value might be sent back by the method, and knowing the parameters allows us to pass the appropriate values into the method to be stored in those parameters.

The details of how the method does its job are unimportant, from the standpoint of calling the method. That is why we don't need to understand, or even see, the code for `Keyboard.readInt()` or `System.out.println()`, or other methods, in order to use them. We simply need to know what to call the method, what parameters we need to supply values for, and what the type is of the value that gets returned, if any value is returned at all. (We generally also need to know what we are accomplishing by calling this method – does this do a calculation? Is is that calculation result that is being returned, or something else, and if so, what? What if we are not doing a calculation; what else does this method accomplish, then? – but usually a small bit of documentation will provide that information.) Given that information, you would be able to write a syntactically-correct method call.

The general form for calling a method – as you have already seen with the methods you have used – is to write the method name, and then in parenthesis list any values being sent to the method.

```
method-name(argument1,...,argument n);
```

The values you send in are called *arguments*.

So, if we want to invoke our `Add3` method, we need to pass in three arguments of type `int`, because `Add3` has three parameters of type `int` that need values. There are many ways we might do this; below is one way, where we send in actual integer literals:

```
Add3(2, 1, 5);
```

We might also have `int` variables `a`, `b`, and `c` that we use as arguments:

```
Add3(a, b, c);
```

or – for any of the arguments – an expression that evalutates to type `int` – since the expression results in a single value:

```
Add3(a, b+1, 2);
Add3(5, c+a, b*a-c);
Add3(  (2-a-b-c)*7,  a+b-c,  6+a/c);
```

We also need to make use of the value returned by this method, so we would call `Add3` the same way we called `Keyboard.readInt()`, namely, as part of an assignment statement.

```
public class Program
{
   public static void main(String[] args)
   {
      int a, b, c, d;
      a = 2;
      b = 1;
      c = 5;
      d = Add3(a, b, c);
          .
          .
          .
```

Completing the method definition

We have the first line of our method definition:

```
int Add3(int x, int y, int z)
```

and now need to finish the definition. First, we need open- and close- braces. Just as we turned a collection of statements into a single compound statement in conditionals and loops by enclosing them in braces, we do the same thing with methods. The difference here is that you *have* to have the braces; even if a method has only one line of code, it needs to be enclosed in a pair of braces.

```
int Add3(int x, int y, int z)
{
   // code goes here
}
```

As far as what code is needed, well, we are trying to add three numbers, and we are pretending that you cannot chain additions together on one line, even though you can, so let's declare what we call a *local variable* to hold the sum, and then progressively add to it.

```
int Add3(int x, int y, int z) {
   int sum;
   sum = x + y;
   sum = sum + z;
}
```

Completing the method definition – return statements

We have calculated the sum, but the last thing we need in our `Add3` method is a way of sending the sum back as the return value of the method. This is done via a *return statement*. The syntax of a return statement is:

```
return expr;
```

where `expr` is some expression (perhaps a non-existent one; more on that later). As we have already seen, an expression can be very complex, or it can be as simple as a single literal or variable. In this case, it will only be a single variable, namely, `sum`.

```
// Final version of this method
int Add3(int x, int y, int z)
{
   int sum;
   sum = x + y;
   sum = sum + z;
   return sum;
}
```

The statement `return sum;` sends back the value of sum as the return value of this method. Since the method is supposed to return a value of type `int`, and since `sum` is indeed a variable of type `int` and thus holds a value of type `int`, everything matches up perfectly.

Overall code (info on next slide)

```
public class Program
{
   public static void main(String[] args)
   {
      int a, b, c, d;
      a = 2;
      b = 1;
      c = 5;
      d = Add3(a, b, c);
      System.out.println("Total is " + d);
   }

   public static int Add3(int x, int y, int z)
   {
      int sum;
      sum = x + y;
      sum = sum + z;
      return sum;
   }
}
```

- Note that "`public static`" appears in front of our new method just like it appears in front of `main()`. The method signature is what you *need* in order to be able to use the method. The `public` and `static` keywords in front of our method definitions are keywords that are there due to some other issues that we will learn about in a few lectures rather than right now. So for now just that assume all methods you write should have `public static` in front of them, and later on we will learn what it means to leave them out or to use different words instead.

- The parameters of `Add3` can be thought of as a special kind of local variable. The local variable `sum` must be declared and assigned inside the braces, but the parameters `x`, `y`, and `z` are "declared" before the open brace is reached, and are also assigned values before the open brace is reached (the values they are assigned, of course, being the values of the arguments sent in to `Add3`). Other than that, they function *exactly* the same way as any local variable such as `sum` that you declare and assign within the braces themselves, and thus you can use those parameters in your code in the same way you would use any local variable that you declared and assigned yourself.

- There is no particular order that the methods need to be in – our example had `main` before `Add3`, but you could have had `Add3` before `main` if you preferred. If the compiler sees a method call before it has encountered that method (as in our example, where the call to `Add3` is seen before the `Add3` method itself), it's not a problem – the compiler just makes a note to itself to keep an eye out for the `Add3` method, and it will complain to you *only* if it has finished trying to compile and has still not found the `Add3` method. If it *does* eventually find the `Add3` method – as it will in our example – then the compiler will make use of that information and will not have any error to report to you in that regard.

- In general, the code inside a method can be as simple or as complex as you want. The code will probably tend to be pretty simple in this class to start with, since we want you to focus on learning method syntax. But usually, very simple code is better left in `main`, and a method should be written for a task that takes more than just one or two lines of code. As with so much else, knowing when to write a method and when not to is a matter of experience, programming style, and the particulars of the situation you are programming for.

Scope in methods

When we went over loops we talked a little about the scope of variables. Specifically, we said that a variable declared within the braces (`{}`) of a compound statement will vanish from existence and no longer be accessible once you have reached the closing brace of that compound statement block.

Methods work in a very similar manner.

- Variables declared within a method have as their scope the life of the method; they vanish when the method ends.
- Other variables in other methods are not accessible from that method. So, if from `main` you invoke `Add3`, you cannot access `main`'s local variables from `Add3`.

Thus, the method scope rule is:

- Variables declared in a method are *only* accessible in *that* method, and in no other methods anywhere else in the program. A method's parameters work the same way as any other local variables of a method and thus go out of scope at the end of the method.

Using methods from other classes

In our program from the last lecture (repeated above), both of the methods were inside the same class. So, `main` was able to invoke `Add3` just by using the name of that method. It is assumed in the absence of any further information that you are talking about the method with that name from the class you are in.

If you want to use methods from another class inside the class you are in, you must list the class name with the method name, so that the system knows in which class to find the method you want. This is done via the following syntax:

```
className.methodName(arg1,...,argN);
```

That is, you list the class, followed by a "dot", followed by the method call as usual. This is the idea behind the call to `Keyboard.readInt()` – the method is called `readInt()` (you will find it if you look inside the
`Keyboard.java` file), and since it is in the class `Keyboard`, you access it by listing the class (`Keyboard`), followed by a dot, followed by the method name (`readInt`).

In the code above, we *could* have written:

```
d = Program.Add3(a, b, c);
```

in place of the method call we do have, but it is not necessary, since `Add3` and `main` are within the same class.

Compilation checks

The compiler does a lot of type checking and other such things when compiling code that uses methods. For each method call, the following things are checked:

1. The name of the method on the calling line must match the name of some actual method that has been written.

2. The number of arguments being sent to this method must match the number of parameters the method has, and their types must match in order – i.e. the type of the first argument must match the type of the first parameter, the type of the second argument must match the type of the second parameter, and so on.

3. If there is a return type other than `void`, then there must be a `return` statement in the method. If the return type is `void`, it's okay to have no `return` statement at all, and it's also okay to have a `return` statement with no expression (i.e. `return;`), which simply exists the method at that point. What you cannot do in this case is have a r̂eturn statement *with* an expression.

4. If there is a return type other than `void` – and thus, by definition, a return statement in the method – then the value of the expression in the return statement – which is the value being returned – must have a type equal to the stated return type. You can't return a `boolean` value from a method with an `int` return type, for example.

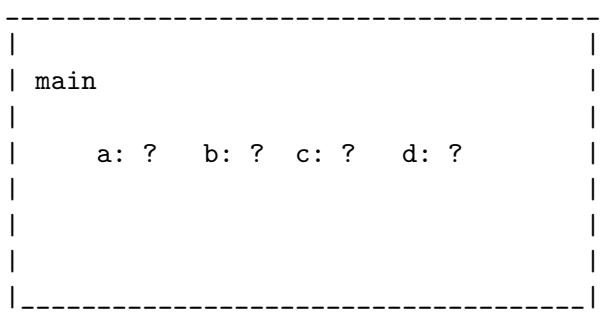
5. The method call statement must be used in a way consistent with its return type. For example, you cannot have a method call to a method that returns void as the right-hand-side of an assignment statement. You cannot have a method call to a method that returns a value of type `boolean` as the right-hand-side of an assignment statement that assigns to a `int` variable. You cannot have as an argument to the logical operator NOT (`!`) a method call that returns `int` (NOT needs a `boolean` value for an argument). And so on.

This makes life very nice for you, because it means that the compiler will catch any syntax or calling/returning consistency mistakes you might have made, rather than somehow having those mistakes crash your program when it runs.

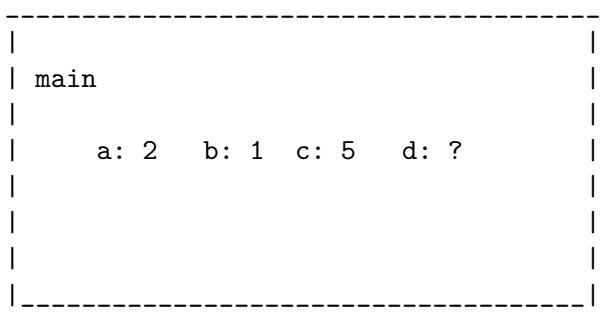
Visual example of method calling

As far as having a visual example of method calling, an analogy to notecards works nicely. This is because, as we have now discussed, the scope for a local variable is that method only – it is not available to any other method. So, we can think of each method call as being on a separate notecard, and the variables declared in that method are unique to that method call – just as if we write some numbers on a notecard, they are unique to that notecard and don't magically appear on other notecards as well.

So, we start off with main (we are again referring to our earlier code example with `main` and `Add3`):

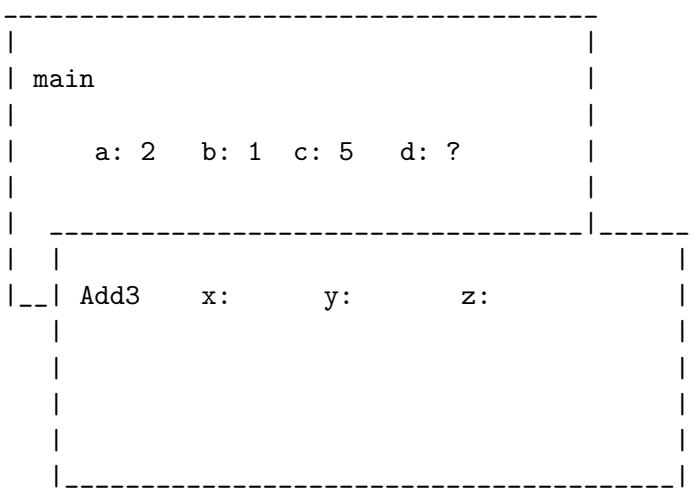


Above we see the `main` method after the declarations have been made. We have made no assignments, so we consider the values in the variables unknown. Next, we will run the three lines of code that assign values to three of the variables.



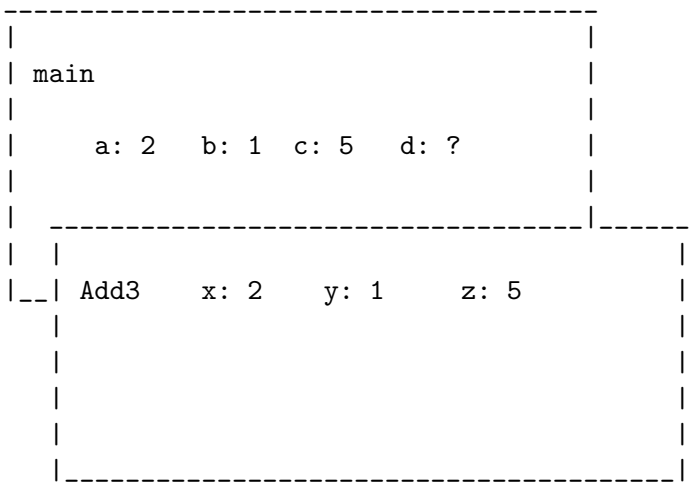
And now we reach the line that assigns to `d`. As always, the expression on the right-hand-side of an assignment statement is evaluated to obtain its value before the actual assignment of that value to the variable can be done. Here, that means we must complete the `Add3` method call before writing the result to `d`, so we next perform the call to `Add3`.

So, let's set up a new notecard for `Add3`. Since we are leaving `main` and thus don't have access to its variables until we return to it, we will place this new notecard *on top of* the one for `main`. In this manner we make it clear that `Add3` (the method on the top notecard) is the currently-active method, and that the methods on notecards below it (in this case, only `main`) are methods we will eventually return to later on as we complete method calls we have begun.

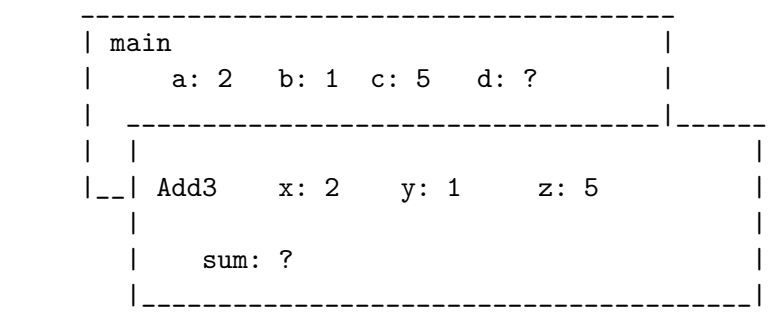


When we first call the method, we have three parameters that need to be assigned values. These parameters are automatically assigned the corresponding values from the method call code line itself. That is, the first argument's value is written into the first parameter, the second argument's value is written into the second parameter, and so on.

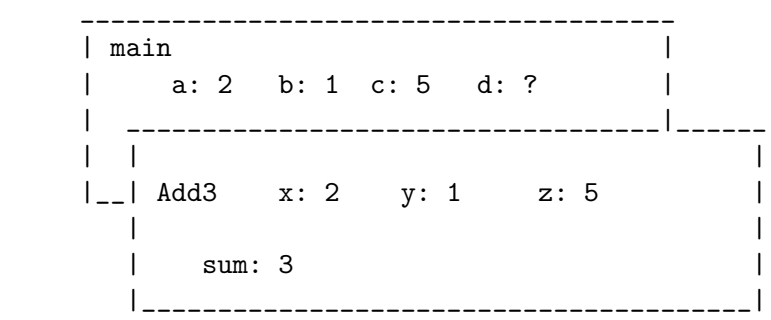
So, above, the value inside `a` is copied into `x`, the value inside `b` is copied into `y`, and the value inside `c` is copied into `z`.



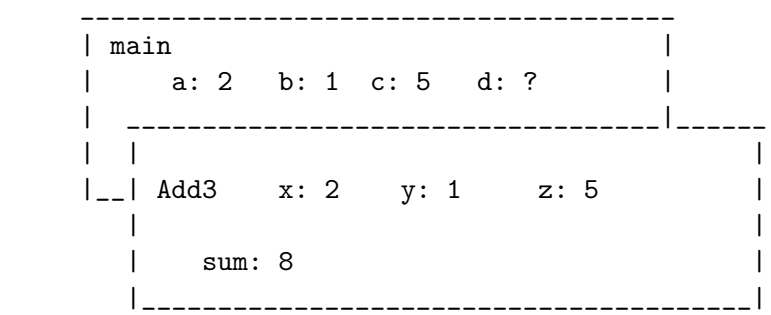
Now, we start the actual code of the `Add3` method. First we declare the local variable `sum` (picture squished to fit it more on slide):



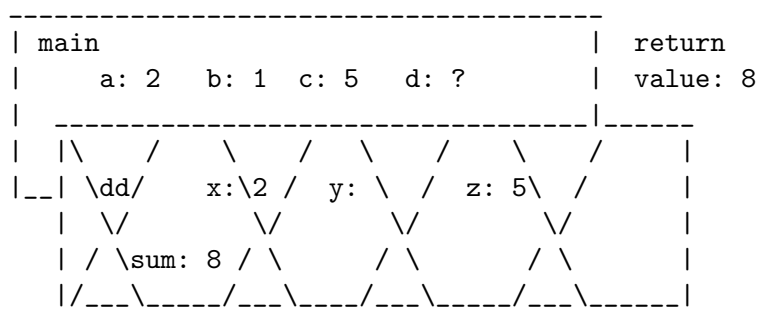
Next, we run `sum = x + y;`...



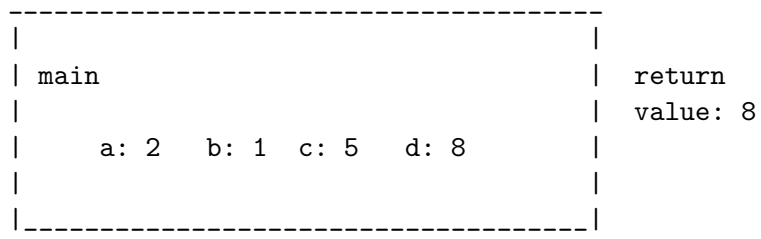
...and then we run `sum = sum + z;`.



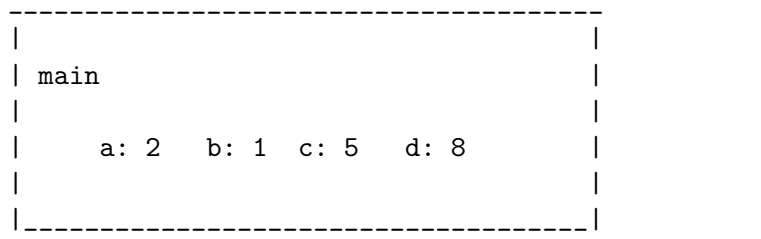
Finally, we come to the statement `return sum;`, which signifies the end of the method. What happens here is that the machine stores the value of `sum` in a "return value" location, and then the method is concluded, meaning that the parameters and local variables go out of scope and thus "vanish". In pictoral terms, this is equivalent to lifing up the notecard and throwing it away.



And finally, the return value gets used in the assignment statement...



...after which the return value itself vanishes (it only gets stored temorarily, until the assignment statement was over).



And now, we are back in `main`, with a value stored inside `d`. However, it is a value we generated using a method call, instead of a value we calculated directly in `main`.

The notecard analogy works especially nicely for methods because methods don't have the ability to communicate with each other freely due to the scoping rules for local variables of methods. Two different method calls really are two separate, distinct processing segments. The only ways that they can communicate are:

1. the calling method can send data to the called method by passing arguments obtained in the calling method to the called method's parameters, and
2. the called method can send data back to the calling method by returning a value (which is why we have a return type).

Structured Programming

The use of methods can lead to a more organized form of program design that serves us better as our programs become larger. Rather than write a complete program as a very long `main()` method, instead we break our large task up into a number of smaller tasks, and we write a method to handle each task. The `main()` method then becomes a "summary" of our program, guiding the overall work, but invoking the various other methods we wrote to handle many of the subproblems involved in the larger task. So, reading over `main()` will still give us a general idea of what is going on in the program, but the details are left to the actual methods that are called from `main()`.

Then, those methods themselves can be broken down in a similar way, where each method makes various *other* method calls to take care of details related to its computation. No one method gets too large, and each method either performs work that is needed, or calls other methods in a certain order to accomplish some work (or both).

This form of programming is known as *structured programming*, because you are taking the large collection of statements in `main()`, and providing structure to that collection by breaking those statements into groups, where each group of statements accomplishes one task.

Another advantage to this technique is that we can "reuse" code statements. Instead of having to write many copies of the same code, we simply write it once, in a method, and then call that method whenever we need to run that code. (That's basically what you've done with your use of `System.out.println()` and `Keyboard.readInt()`.) One big advantage to this is that if we need to make changes to the code in question, we only need to change it once, in the method, and not in many places all over the program. If we did not have methods, and instead pasted the same code into many different places in the program, then any change to that code would have to be made in all the different places where it appeared, rather than just changing it once in the method.

This organization also makes a program easier to read, especially if you are new to the program and are seeing the code for the first time, because you can read the "overview" of the code and get a basic idea of how the program runs, without having to inspect every detail of every method. (For example, you did not need to understand how `Keyboard.readInt()` worked in order to use it.) As long as the method names are descriptive and there is good documentation about what those methods do, there should be little to no reason to have to inspect the details of every method simply to understand how the code works.

So structured programming has a lot of advantages over the "put all your code in `main()`" approach to program design.