Methods

We now have a number of tools for processing data and making complex decisions. Our programs can now get pretty complicated, which requires a lot of thinking to keep track of how they work. As good computer scientists, we should be starting to feel worried that we need to be lazier, we need tools to help us organize our code into small pieces so that we never have to keep the whole program in mind at once.

Also, we often write a multi-step task that we then need to apply to various different data values. We could do this by copy-pasting the code and putting in different specific values, but any time we have multiple copies of the same code, that means we have to be responsible for keeping track of those copies and updating all of them any time we need to adjust how they work… which we should be too lazy to do.

The solution to both of these problems is to divide our programs up into methods. Methods are also known in other languages as modules, functions, procedures, subroutines, etc.

Method Header

Each method begins with a header that gives it a name, says what values it needs as input from the rest of the program, and says what type of result it will give back to the rest of the program.

```
returnType methodName(type1 param1, type2 param2, …) {
```

The method name follows the usual rules for naming and should indicate what the code inside the method body will do.

The parameters are a list of zero or more local variables that the method will use to perform this code. Each parameter will start with a value that it is given by another part of the program when the method runs. We don't know what those values will be before the method runs, so we have to write the method so that it can deal correctly with *any* possible value based on the type.

The return type is the type of a value that the method will give back to another part of the program when it is finished. This can be any valid value for a variable. The return type could also be the special type *void* which means that the method does not return any value at all – it just does a series of steps but does not give back a result.

To start off with, we'll create very simple methods that have zero parameters and type void, so they get no values from the rest of the program, and give back no result.

Method Body

Like other structures, a method can have code inside it, which is called its body. We use curly braces {} to begin and end the method's body.

```
void printHello5() {
    //body of the method
    for (int i = 0; i < 5; i=i+1) {
        print(“Hello”);
     }
} // end of the method
```

The body of a method can include any code we would put anywhere else including conditionals, loops, printing, etc, but we would never create a method inside another method.

Local Variables

Variables created inside a method are local to that method – they only exist within that method (“in that method's scope”), while it is running. They do not persist afterwards. This means that multiple methods could have variables with the same name without interfering with each other.

Consider readability when choosing local variable names; if two methods are doing related things to the same kind of value, it may make more sense for them to use the same variable name. If two methods are doing very different things, it may cut down on confusion if they use different variable names as well.

Calling Methods

The point of a method is that it is a tool that other parts of the code can use. Methods are used by *calling*, that is, using their name. The name of a method always includes the parentheses after it, with values for any parameters inside.

In many languages, if a program has multiple methods, it should have one method called main(), a.k.a. the main method. The program automatically begins with the first line of main, and if main calls another method, that method is run at that point in main. If main does not call a method, it does not run, even if it is present in the code.

```
1   void square() {
2       int userNum = prompt("your number? ")
3       print("square is " + userNum * userNum)
```

```
4   }
5
6   void cube() {
7       int userNum = prompt("your number? ")
8       print ("cube is "
9            + userNum * userNum * userNum)
10  }
11
12  void main() {
13      int user = prompt ("1 for cube, 2 for square ")
14      if (user == 1) {
15          cube(
16      } else {
17          square()
18      }
19
20      print("Goodbye")
21  }
```

In this code, the first thing that happens is that the user is prompted to choose between cube or square in main. If the user chooses 1, the cube() method runs. Once the cube method finishes, we are back in main, and we print "Goodbye" and in that case, the square() method never runs. if the user chooses anything else, the square method runs, then we print "Goodbye" but in that case the cube() method never runs.

In this example, I put square() and cube() above main(). In some languages, the order of methods is important because you cannot call a method until after you have defined what the method does. In other languages, the order of whole methods does not matter. The code *inside* each method is still run from top to bottom in any case. In this example I put main at the bottom, but remember that the program will start with the first line inside the main, no matter where main is located in relation to other methods.

The Stack

When a method is called, a chunk of main memory space for the program is taken up by the method, including space for any local variables it has. The part of program memory used for methods is called the *stack* and each method's chunk of space is called a *stack frame*.

When the method ends, the stack frame is "given back" that is, that chunk of main memory space is marked as no longer occupied; it is open for use to store a new stack frame for another method. If one method calls another which calls another, then we could end up with multiple stackframes which take a while before they come back off the stack.

Parameters

Reading input from a user (or file, or…) is one way to get input into a method. To make methods even more useful, we need a way to get input from the part of the program where the method is called into the method, so we can do the same code multiple times but with different starting values without having to get each one from outside the program. We do this by *passing* values into the method's parameters.

Parameter Lists

Each method is defined with a list of zero or more parameters in its parentheses. These are local variables in the method, but unlike variables created in the body of the method, they are automatically given values before the method starts. When writing the method, you don't give them values, but you do have to think about how to *write the method so that it will work properly no matter what value the parameters start with*. The method should be a general tool that will work whatever values the parameters start with

Like any other variable, each parameter needs a type, and any type valid for variables can be used for parameter types.

If you need the variable to start with one specific value, then it should not be a parameter, it should be a local variable defined in the body of the method. If you need input from the user, the variable for that should not be a parameter, it should be a local variable defined in the body of the method. The point of a parameter is that it is a way for *another part of the program* to feed a value to the method.

When we talk about method parameters, we use words like "takes" "needs" "starts with" or "given" while if we were reading the value of a local variable as user input, we would say "takes as input" "asks the user for" "takes as input" "given by the user".

Calling Methods with Parameters

Parameters to a method are listed in a specific order. When the method is called, it must be given values in the same order, in its parentheses. Those values might be literals, or stored in variables, or creating by evaluating an expression. We might talk about calling a

method with given values in its parentheses as calling it “of” “on” “with” those values or “applied to” those values.

In this example, two methods each have two parameters.

```
1   // either do or don’t print the given string
2   void printDo(boolean printIt, string word) {
3       if (printIt) {
4           print(word)
5       }
6   }
7
8   // print the given string a number of times
9   void printTimes(string word, int howMany) {
10      for (int i = 0; i < howMany; i=i+1) {
11          print(word)
12      }
13  }
14
15  void main(string[] args) {
16      printTimes("hello", 3)
17      printDo(true, "hi")
18
19      int times = 2
20      printTimes("goodbye", times)
21      printDo(true, "bye")
22
23      string str = "this is awkward"
24      printTimes(str, - 3)
25      printDo(false, "huh")
26
27      print("END")
28  }
```

Here is the output we would see when the above code runs annotated with the lines causing the output

```
hello                line 16 / line 11
hello                line 16 / line 11
hello                line 16 / line 11
hi                   line 17 / line 4
goodbye              line 20 / line 11
goodbye         line 20 / line 11
bye                  line 21 / line 4
END                  line 27
```

While line 16 runs, memory would look like this (note the counter i is counting its way up to howMany)

variable	*MM*	*value*
[stackframe: main]		
	50	
	51	
	...	
[stackframe:printTimes("hello", 3)]		
word	60	"hello"
howMany	61	3
i	62	~~0~~ ~~1~~ ~~2~~ 3
	...	

After this, the stackframe for printTimes would be given back, so the next method call could re-use the same space, so while line 17 runs, memory could look like

variable	*MM*	*value*
[stackframe: main (args)]		
	50	
	51	
	...	
[stackframe:printDo(true, "hi")]		
printIt	60	true
word	61	"hi"
	...	

While line 20 runs (notice that howMany gets the value of the times variable, 2:

variable	*MM*	*value*

[stackframe: main ()]		
times	50	2
	51	
	…	
[stackframe:printTimes(“goodbye”, times)]		
word	60	“goodbye”
howMany	61	2
i	62	~~0~~ 1 2
	…	

While line 24 runs (this time word gets the value of the variable str and howMany gets the value of the expression times – 3

variable	*MM*	*value*
[stackframe: main ()]		
times	50	2
str	51	“this is awkward”
	52	
	…	
[stackframe:printTimes(str, times - 3)]		
word	60	“this is awkward
howMany	61	3
i	62	0
	…	

Main calls each method three times with different values for the parameters, called *arguments*. (I will only talk about the stack for the first call, all the others are similar).

- printTimes(“hello”, 3)
    - just before printTimes begins, a stack frame is placed on the stack and in it are space for the parameter variable word, which is initialized to the value “hello” and the parameter variable howMany, which is initialized to the value 3
    - the method runs and “hello” is printed three times
    - the method ends, and we give back the stack frame, so this space in main memory can be re-used for other methods. Now we are back in main()
- printDo(true, “hi”)
    - just before printDo begins, the parameter variable printIt is initialized to the value true and the parameter variable howMany is initialized to the value “hi”
    - the method runs, printing “hi”

    - the method ends and we are back in main()
- printTimes("goodbye", times)
    - just before printTimes begins, the parameter variable word is initialized to the value "goodbye" and the parameter variable howMany is initialized to the value 2 (the values of the times variable)
        - printTimes does *not* have access to the times variable itself; it would *never* make sense to try to use the times variale in printTimes. We would never write printTimes to be dependent on the name of a variable from main!
    - the method runs and "goodbye" is printed two times
    - the method ends, and we are back in main()
- printDo(true, "bye")
    - just before printDo begins, the parameter variable printIt is initialized to the value true and the parameter variable howMany is initialized to the value "bye"
    - the method runs, printing "bye"
    - the method ends and we are back in main()
- printTimes(str, times - 3)
    - just before printTimes begins, the parameter variable word is initialized to the value "this is awkward" and the parameter variable howMany is initialized to the value -1 (the result of subtracting 3 from the value of the times variable)
    - the method runs and since the for loop starts at 0 and goes only as long as it is < howMany, which is -1, nothing is printed
    - the method ends, and we are back in main()
- printDo(false, "huh")
    - just before printDo begins, the parameter variable printIt is initialized to the value false and the parameter variable howMany is initialized to the value "huh"
    - the method runs, but since it checks printIt, which is false, it doesn't print anything
    - the method ends and we are back in main()

Notice that we were able to pass in literal values, values stored in variables, and values from expressions as our arguments.

In general, the reason we are passing values into a method is so that the method can use those values, so in most cases we wouldn't be giving a parameter variable a new value inside the method body, but we might *adjust* the value of a parameter in the course of the

task the method does, so it is worth considering what happens when we pass a variable in as argument to a parameter that is changed by its method.

```
1   void countDownFrom(int top) {
2        // since top is already initialized
3        // there is no initialization step
4        // in this for loop - weird but allowed!
5        for (          ; top > 0; top = top - 1) {
6            print(top)
7        }
8        print("YAY")
9    }
10
11   void main() {
12       countDownFrom(10)
13
14       int k = 5
15       countDownFrom(k)
16
17       print("k is now " + k)
18   }
```

While line 12 runs, memory might look like this:

variable	*MM*	*value*
[stackframe: main ()]		
	50	
	...	
[stackframe:countDownFrom(10)]		
top	60	~~10~~ ~~9~~ ~~8~~ ~~7~~ ~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0
	...	

While line 15 runs, memory might look like this:

variable	*MM*	*value*
[stackframe: main ()]		
k	50	5
	51	
	...	
[stackframe:countDownFrom(k)]		
top	60	~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0
	...	

In this code, the countDownFrom method uses the top parameter as its counter, counting down from whatever value top was initialized to until top is down to 0, decreasing each time.

When we call countDownFrom(10), the top parameter variable is initialized to 10, and then it counts down from 10, changing the top variable each time through the loop. There was no variable in main, so there is nothing to change.

When we call countDownFrom(k), the top parameter variable is initialized to 5, which is the value of k from main. The code counts down from 5, each time through the loop changing the value of top. But does this also change the value of k?

Approaches to Arguments

The most common approach is *pass by value*: a parameter is initialized with a copy of the value that is given when the method is called. In this case, any change made inside the method would not affect any variable that was passed in for that parameter.

In this case, no change made to a parameter's value inside the called method can change the value of a variable in the calling method. In our example above, top would be initialized to 5, a copy of the value of k, but the changes made to k after that would not affect k, and k would still be 5 after the countDownFrom method finished for the second time.

Another approach is *pass by reference*: a parameter is assigned the same memory address as a variable passed in as argument. So the parameter variable becomes another name for the same storage location in main memory – two names for the same thing – and either variable name can be used to access that spot in main memory. So any changes made to the parameter variable are immediately made to the value stored by the original variable. When we learn about reference variables (a.k.a. pointers) more generally we will have a better sense for how this works.

For the moment, we will assume that arrays and only arrays are passed by reference. Remember that an array variable is actually the first address in a block of addresses in main memory and the index is an offset saying how far down in that block to go. So if we pass an array into a method, the method will have access to that same block of main memory. This means that changes made to the elements of an array during a method *will* persist after the method ends, because it is the same array, not just a copy.

```
void addList(int list[], int add){
   for (int i = 0; i < list.size; i = i + 1)
         list[i] = list[i] + add
```

```
    }
}

void main(){
    int numList[5] = {2, 4, 6, 8, 10}
    addList(numList, 1)
    print "0th element is now " + numlist[0]
}
```

Since numList is an array, when it is passed into addList, the parameter variable list isn't just a copy of the array, it is the same array. So, when the addList method adds a value to each element of the list array, numList's values are changed (because they are the same) and when we get back to main, we will see that numList's 0^{th} value is now 3, not 2.

There are also less common approaches to passing variables that are used in some languages, but pass by value and pass by reference are how most languages handle parameters.

Returning from Methods

Just as we use parameters to get values from the caller into a method, a method can give a value back to a caller by returning it.

Return Type

Any type that is valid for a variable is also valid for a method's return type, including class types. We also have the special type `void` to indicate that a method does not return anything. If a method specifies a return type other than void, the method must return a value of that type by the end of the method. A method can return any value of its type, including values like zero or null. Only a single value can be returned (that single value could be the address of the 0^{th} element of an array of values).

Return

The keyword return is followed by the value (matching the return type) that this method will evaluate to. It could return any expression that evaluates to the correct type, including a literal value, a variable, or the result of doing operations

```
double sumDouble(double a, double b) {
    double sd = (a + b) * 2
    // return statement
    return sd;
```

```
}
```

In the example above, we did the math in the expression and put the result into a variable, then returned the variable. It is somewhat more common, in a case like this, just to put the expression after the keyword return:

```
double sumDouble(double a, double b) {
   return (a + b) * 2
}
```

A return statement ends the method, so a method can't have any statements after a return. For this reason, we should never have a return inside a loop unless it is also in a conditional that means it only happens under certain circumstances (which mean the method should end early at that point).

Many people feel that code is easier to read and debug if there is only one return statement per method, so they would use a variable to store the eventual value to return, change that variable as needed with conditionals, and then return that variable at the end. In more complex situations we might have several local variables that get set and then combined at the end for a return value.

This works well if the method always returns something of the same format (a string concatenated in the same final order, or a numeric value calculated using the same final formula). It also makes life easier if there is some series of steps that must happen before the method ends, sometimes thought of as "cleanup." If the method has been using a resource such as a file or network connection, we probably need to clean up that connection before the method ends, which is easier if the method has just one ending.

As with many rules of thumb for making code simpler, this rule is about avoiding common mistakes that make code hard to read and maintain. In a case where a method has to return very different kinds of values in different situations, or sometimes gets to its result in a long complex process but in other cases ends very early, you might find that twisting your code to fit it to a single return actually makes it harder to read. A single return is not a bad guideline, but making your code express the underlying situation clearly is more important.

Even if we write a method using multiple return statements, only one of them can happen, because as soon as that return statement runs, the method ends.

Here is an example done first with two returns, then with one.

```
int findPosition(string [] namelist, string find){
   for(int i = 0; i < namelist.size; i = i + 1) {
```

```
            // if we find the name at the position
            // return the position
            // this ends the loop early
            if (namelist[i] == find) {
                 return i
            }
    }
    // if we got all the way to the end without finding it
    // use -1 to indicate not found
    return -1
}

int findPosition(string [] namelist, string find){
    int pos = -1 // impossible value
    // adjust condition of for loop so we stop early
    // if the position variable has changed
    for(int i = 0; i < namelist.size AND pos == -1; i = i + 1) {
            // if we find the name at the position
            // return the position
            // this ends the loop early
            if (namelist[i] == find) {
                 pos = i
            }
    }
    return pos
}
```

When we talk about methods that return, we would say things like it "returns" "gives back" or "results in" the value, while to say that it prints instead, we would say it "reports" "tells" the user" or "shows".

Calling Methods that Return

When we call a method that returns, that method call evaluates to the returned value – imagine the method call turning into the return value. Most of the time, if we called a method that returns, it is because we want to do something with that value. So although it is legal to just call a method that returns as a statement of its own, this is usually a bad idea. We should be storing the return value in a variable or else using it in another expression, maybe passing it to another method (say, to print it).

If we fail to do this, we sometimes say that we “dropped the return value on the floor” imagining that the method tossed its return value back to our code and we failed to catch it.

```
double sumDouble(double a, double b) {
    return (a + b) * 2
}

void main(string[] args) {
    // useless, we drop 14 on the floor
    sumDouble(5.1, 1.9)

    // catching 14 in a variable
    double result = sumDouble(5.1, 1.9)

    // printing 14
    print("your result is " + sumDouble(5.1, 1.9))

    // using return values as arguments
    double bigMath =
        sumDouble(sumDouble(2.2, 3.3), sumDouble(1.1, 5.5))
        // evaluated as:
        // sumDouble(11.0, sumDouble(1.1, 5.5))
        // sumDouble(11.0, 13.2)
        // 48.8
    print("result of big math was " + bigMath)
}
```

So at the end, we print “result of big math was 48.8”

Recursion

Suppose we have two methods, methodA and methodB, and into methodA we put a call to methodB, but then into methodB we put a call to methodA. Then methodA would call methodB which would call methodA which would call methodB which would call method…

This is called *mutual recursion*.

If into methodA we put a call to methodA itself, then this is just *recursion*.

Certain problems have recursive definitions, in which case it makes sense to write recursive methods to solve them. For example, the nth Fibonacci number is defined as the sum of the n-1st and n-2nd Fibonacci numbers. To write this as code:

```
int fibonacci(int n){
       return fibonacci(n-1) + fibonacci (n-2)
}
```

There is a problem here, however. This would just keep running forever; we never told it where to stop. Actually, the Fibonacci numbers are only defined this way as far back as the 2nd Fibonacci number. The 0th and 1st Fibonacci numbers are just defined as 1.

For recursion to be useful, like a loop, it has to have an ending point. This is called the *base case*. So a recursive method should start with a conditional that checks whether we are in a base case, in which case it does whatever the base case does.

Otherwise, we can do the *recursive call* which means calling the method itself, but on a new argument that is closer to the base case than the argument we were given.

```
int fibonacci(int n)
   // check for base case
   if (n <= 1) then
        return 1
   endif
   // recursive call (n-1 and n-2 are closer to the base case)
  return fibonacci(n-1) + fibonacci (n-2)
end fibonacci
```

Remember that every time we call a method, that method gets space on the stack. Each time we do this takes up time and space. We end up getting an error called a *stack overflow* if we put too many method calls on the stack; and even if not, it may take a very long time to work through all those method calls, even if each one is very simple, because of the extra time spent just on setting up and cleaning up the stack frame.

So, in most languages it is much faster and more space efficient to use loops instead of recursion to repeat code, except when we need the form of the code to reflect the inherent recursive nature of our understanding of a task.

Overloading

Choosing good names for methods is an important programming skill; we want the name to clearly communicate what the method does so that when we read a single statement that calls the method, we have good expectations for that many lines of code inside the method that this may stand for.

Sometimes we need to write several methods that do the same task, but the parameters are different in number or type. For readability, it may actually be better to give all these methods the same name. When we have multiple names with the same name but different parameters, this is called *overloading* a method.

Operators like + or AND are a tool in a program that act on values they are given and result in a new value. So operators are very much like methods. In some languages, operators *are* methods (with unusual syntax) and they can be overloaded, so we can say, for instance, that in addition to what + means for numbers (addition) and what + means for strings (concatenation) we might define what + means for other types. In our pseudocode we assume that any other type + string is defined (to cast the other type to a string and then do concatenation. But we might decide to also choose what * means if we put it between a number and a string. This can be very useful in languages where programmers can define their own types. We may also be able to define our own operators using other symbols.

Global Variables

So far, our variables have always been local, living inside a single method (often our whole program was one big method we didn't bother to name). Now each of our methods can define its own variables, but each such local variable only exists in that method while it is running.

It is often tempting to say that we would like to create a single variable that is accessible in all our methods, so they can all share it without having to constantly pass it in as parameter and return it. Such a variable is called a *global variable*.

```
double balance = 0

void deposit(){
   double user = prompt("how much?")
   balance = balance + user
}
```

```
void withdraw(){
   double user = prompt("how much?")
   balance = balance - user
}

void menu(){
   int user = -1
   constant int QUIT = 0
   while (user != QUIT){
         print(“1 to see balance”)
         print(“2 to deposit”)
         print(“3 to withdraw”)
         print (QUIT + “ to quit”)
         user = prompt(“Choice?”)
         case user{
              1: print(“$” + balance)
              2: deposit()
              3: withdraw()
         }
   }
}

void main(){
   menu()
   print("Final Balance: $" + balance)
}
```

All the methods in this code had access to the balance variable. Some changed it. These methods were designed to be used with the balance. But every other method we added to the code would also have access to the balance, and would be able to change it. The more methods, the more complex the situation gets.

Global variables are one of those coding practices that make code superficially easier to write (especially if you are intimidated by passing in parameters and returning values) but in practice often makes code harder to read, debug, and maintain. To use them without problems takes careful design.

Many programmers avoid all global variables except for constants, which gives us the advantage of creating named constants for readability that only have to be defined once and are available throughout code – so we are consistent about using the same value

everywhere, but since they are constant, we never have to try to chase down which part of the code keeps messing up their value.

In object oriented design we have the idea of an instance variable, which gets us get some of the advantages of global variables while maintaining our future laziness by putting a limit on complexity. An instance variable is available to all methods within a specific code scope called a class. The instance variable represents a permanent characteristic that influences and may be changed by this specific set of methods, which represent behaviors of whatever object has that characteristic. The methods and the instance variable are designed to work together. But the instance variable is not available outside that scope and we have structured ways to control access to it.