Methods

Why Write Methods?

Many times when we are writing programs we find that we want to repeat the same set of actions multiple times. When we put all the code in a single Main method, we must just write the same lines of code over and over. This leads to programs that are longer than they need to be and "ugly" code.

A *method* is a section of code that does a particular job. When we want to execute those lines of code, we *call* the method. This means that instead of repeating a bunch of lines of code when we want to perform an action, we just need to include one line to call the appropriate method. This makes our code much easier to read.

Declaring a Method

This section will discuss the format for declaring a method. All methods must have a *return type* (if the method is performing a calculation and giving back a result), a *name*, and a number of *parameters* (pieces of data that the method needs to perform its calculations).

Syntax

Methods are declared inside the brackets for the class, but outside the brackets for the main method. For example:

```
public class Example {
     public static void main(String[] args) {

     }

     //Declare method here

     //Can declare as many methods as you want
}
```

Here is the syntax for declaring a method:

```
public static returnType name(params) {
     //code for method goes here
}
```

For now, don't worry about what the **`public static`** part means – just put that at the beginning of all your methods.

Next, the `returnType` is the type of data you want this method to give back. For example, if we were writing a method that computed the area of a square with integer sides, then we would

want to return an `int` – the data type of the area. If you do not want your method to give back a result, then the return type should be **`void`**.

The `name` of method should be descriptive of what the method is doing. For example, a method that computes the area of a square should be called something like `area` or `areaSquare`. The rules for naming methods are the same as the rules for naming variables: method names can be made up of letters, numbers, and underscores, but they cannot start with a number.

Finally, the `params` are a comma-separated list of values that the method needs to do its calculations. Each parameter is listed with a data type and a name. If the method does not need any values to do its calculations, then the `params` section is left off the method declaration (but the `()` at the end of the method is still there).

Examples
Suppose we want to write a method that computes and returns the area of a rectangle. First, let's think about what information this method needs to do its computation – these will be the parameters. To compute the area of a rectangle, we need its width and its height. Next, we need to determine the data type of these values. We know that the width and height will be numbers, but they can be either ints or doubles – depending on what kind of values we want to allow. We'll be more general and make them doubles.

Next, we need to consider what kind of value we're computing. We want to return the area of a rectangle. Each side of the rectangle is a double, and so the area (and return type) should also be a double.

Here is the declaration for the method that will compute the area of a rectangle:

```
// This method returns the area of the rectangle with
// length  length and width width
public static double area(double length, double width) {

}
```

As a second example, suppose we want to write a method that prints all of the values in an array of integers. This method needs to take in the integer array as a parameter, but it doesn't need to return anything (so it will have a `void` return type). Here is the declaration:

```
//This method prints every value in the  nums array
public static void printArray(int[] nums) {

}
```

Finally, suppose we want to write a method that returns the number of times a particular character appears in a string. This method needs to take the string we're examining and the letter

we're looking for as parameters. It needs to return a count of how often that letter is in the string – an int. Here is the declaration:

```
//This method returns the number of times letter appears in str
public static int countLetter(String str, char letter) {

}
```

Writing a Method

Now that we have practiced declaring a method, we are ready to fill in the code for methods. This code will look very similar to things we have been writing inside our `main` method (after all, `main` is also a method).

Void Methods

We will first look at how to write the code for a void method (a method that does not return a value). These methods are slightly easier to implement than methods that return values.

Recall the method declaration for printing all the values in an integer array:

```
//This method prints every value in the nums array
public static void printArray(int[] nums) {

}
```

Now, we want to write code inside the brackets (`{ }`) of the method that will print all values in the `nums` array. This works exactly like it would if we were writing it in the `Main` method – we just treat `nums` like we would any other variable. Here is the complete method:

```
//This method prints every value in the nums array
public static void printArray(int[] nums) {
     for (int i = 0; i < nums.length; i++) {
          System.out.println(nums[i]);
     }
}
```

For another example, suppose we want to write a method that asks the user to input a certain number of integers, and then prints the sum of those numbers. This method needs to take the number of elements we want as an int parameter. It does not need to return anything because it is printing its result. Here is the complete method:

```
//This method asks the user for n numbers,
//and prints the sum of those numbers
public static void sumN(int n) {
     int sum = 0;
```

```
        Scanner s = new Scanner(System.in);

        for (int i = 0; i < n; i++) {
             System.out.print("Enter an integer: ");
             int val = Integer.parseInt(s.nextLine());
             sum += val;
        }

        System.out.println("The sum is: " + sum);
    }
```

Methods that Return a Value

Methods that return a value look fairly similar to void methods. The only difference is that methods that return a value must have the statement:

```
return expression;
```

where `expression` is either the name of a variable, some expression (a math operation, for example), or a constant value (like 4, true, or "hello"). Furthermore, expression should have the same data type as the return type for the method.

Recall the method declaration for computing the area of a rectangle:

```
// This method returns the area of the rectangle with
// length  length and width width
public static double area(double length, double width) {

}
```

The formula for the area of a rectangle is the length times the width, so the completed method looks like this:

```
// This method returns the area of the rectangle with
// length  length and width width
public static double area(double length, double width) {
     return length*width;
}
```

Notice that `length*width` is an expression whose data type evaluates to a double – the same type as the return type.

Next, recall the method declaration for counting the occurrences of a character in a string:

```
//This method returns the number of times letter appears in str
public static int countLetter(String str, char letter) {
```

```
}
```

To implement this method, we will have to loop through all the characters in `str`, and add one to a count variable every time a character matches `letter`. After we have finished looping through the string, we will return our count (which should have an int data type). Here is the implementation:

```
//This method returns the number of times letter appears in str
public static int countLetter(String str, char letter) {
     int count = 0;
     for (int i = 0; i < str.length(); i++) {
          if (str.charAt(i) == letter) count++;
     }

     return count;
}
```

Finally, suppose we want to write a method that determines whether or not a particular letter is a character in a string. This method will again need to take the string and the character as parameters. Since it is computing whether or not something is true, its return type should be a `boolean`. Here is the complete method:

```
//This method returns true if letter is in str, and false otherwise
public static boolean containsLetter(String str, char letter) {
     bool foundIt = false;
     for (int i = 0; i < str.length(); i++) {
          if (str.charAt(i) == letter) foundIt = true;
     }

     return foundIt;
}
```

This method is not as efficient as it could be, though. Suppose the first character in `str` matches `letter` – do we really need to look through the rest of the string? Here is an optimized version of the same method:

```
//This method returns true if letter is in str, and false otherwise
public static boolean containsLetter(String str, char letter) {
     for (int i = 0; i < str.length(); i++) {
          //We've found it – no need to continue looking
          if (str.charAt(i) == letter) return true;
     }

     //We must not have found it, or we would have already returned
     return false;
}
```

Notice that in this version, there is a return statement in the middle of the method. That's OK – it just means that if we do find a matching character, we immediately leave the method and return true without checking any other characters. If we do happen to make it out of the loop, we must not have found a match (otherwise we would have already returned true). So at this point, we can return false.

Rules for Returning Values

There are several rules for how to return values in non-void methods:

- Every possible path through the method must end in a return statement
- Consequently, a path cannot have any other code after its return statement

Let's look at a couple of methods and determine whether or not they follow these rules:

```
//Incorrect: no return statement if num1 != num2
//This method returns whether num1 equals num2
public static boolean equal(int num1, int num2) {
     if (num1 == num2) return true;
}
```

The above method is not correct. If `num1` does not equal `num2`, then no value will be returned. There must be a return statement for every possible scenario.

```
//Incorrect: print statement will never be reached
//This method returns whether num1 equals num2
public static boolean equal(int num1, int num2) {
     if (num1 == num2) return true;
     else {
          return false;
          System.out.println("Not equal");
     }
}
```

The above method is not correct. If `num1` does not equal `num2`, we will return false. A return statement forces us to leave the method, so the print statement will never be executed. While this wouldn't cause any major problems, it will give you a compiler error.

```
//Correct
//This method returns whether num1 equals num2
public static boolean equal(int num1, int num2) {
     if (num1 == num2) return true;
     else return true;
}
```

The above method is correct. If `num1` equals `num2`, we return true. If `num1` does not equal `num2`, we return false. Those are the only possible scenarios, and each of them has a return statement. We also don't try to do anything else after returning a value.

Calling Methods

Now that we can write separate methods, we need to be able to *call* them (tell the compiler that we want to execute the code in the method). Suppose our program looks like this, which includes some of the methods written above and some new ones:

```
public class Methods {
     public static void main(String[] args) {
          //The code we write in this section goes here
     }

     //This method prints every value in the  nums array
     public static void printArray(int[] nums) {
          for (int i = 0; i < nums.length; i++) {
               System.out.println(nums[i]);
          }
     }

     //This method returns the number of times letter appears in str
     public static int countLetter(String str, char letter) {
          int count = 0;
          for (int i = 0; i < str.length(); i++) {
               if (str.charAt(i) == letter) count++;
          }

          return count;
     }

     // This method returns the area of the rectangle with
     // length  length and width width
     public static double area(double length, double width) {
          return length*width;
     }

     //This method asks the user for 10 numbers,
     //and prints the sum of those numbers
     public static void sum10() {
          int sum = 0;
          Scanner s = new Scanner(System.in);

          for (int i = 0; i < n; i++) {
```

```
            System.out.print("Enter an integer: ");
            int val = Integer.parseInt(s.nextLine());
            sum += val;
        }

        System.out.println("The sum is: " + sum);
    }
}
```

Calling a Void Method
Here is the syntax for calling a method with a `void` return type:

```
ClassName.methodName(params);
```

Here, `ClassName` is the name of the class that we're using, and `methodName` is the name of the method that we want to run. Finally, `params` is a comma-separated list of values that you want to pass to the method. Each value can be a variable name, a constant value, or an expression (like a math operation). You do not list the type of the parameters when you call the method, but their types should match the parameter types in the method declaration.

Suppose we are inside the `main` method in the `Methods` class above. Let's say we want to call the `printArray` method. First, we create an array of integers:

```
//creates space for the array and initializes the values
int[] vals = {1, 2, 3, 4};
```

Now, we are ready to call the `printArray` method. Here's how:

```
Methods.printArray(vals);
```

Notice that we just list "`vals`" as the name of the parameter. This variable has the same type as the parameter in the `printArray` method declaration – an `int[]`.

When we call the `printArray` method, we pass the `vals` array. The `printArray` parameter is named `nums`, so `nums` gets initialized to be this `vals` array. When we print the elements in `nums` inside the method, it is really printing the elements from `vals`.

Suppose we are still inside the `main` method in the `Methods` class, and that we now want to call the `sum10` method. Here's how:

```
Methods.sum10();
```

This method does not take any parameters, but we still end the method call with `()`. Now, our entire `main` method looks like this:

```
int[] vals = {1, 2, 3, 4};
Methods.printArray(vals);
Methods.sum10();
```

When our program is executed, it starts with the very first statement in the `main` method. This statement creates our `vals` array. Next, we execute the call to the `printArray` method. This method prints every element in our array. When the method finishes, the program returns back to the spot in the code where the method was called. So the very next thing that we do is execute the call to the `sum10` method. This asks the user for 10 different numbers, and prints their sum. When this method finishes, it returns back to the `main` method. That's the end of the `main` method, so our program ends.

Calling a Non-Void Method

Calling a method that returns a value is a bit trickier. For now, the syntax for calling one of these methods is:

```
type name = ClassName.methodName(params);
```

Here, we are declaring a variable called `name` and setting it equal to the result of the method call. This will store the value returned by the method in the `name` variable. The type of the `name` variable must match the return type of the method.

For example, suppose we are in the `main` method in the `Methods` class, and we want to compute the area of a 3x4 rectangle. Here's how:

```
int result = Methods.area(3.0, 4.0);
```

When this method is called, `3.0` is passed for the `length` parameter, and `4.0` is passed for the `width` parameter. The method returns the calculation `3.0 * 4.0`, which gets stored in our `result` variable.

Next, suppose we want to call the `countLetter` method from the `main` method in the `Methods` class. Suppose that we first want to ask the user to input a string and character to use, and then we want to give those values to the `countLetter` method. Here's how:

```
Scanner s = new Scanner(System.in);
System.out.print("Enter a word: ")
String word = s.nextLine();
System.out.print("Enter a letter to search for: ");
char c = (s.nextLine()).charAt(0);

int numTimes = Methods.countLetter(word, c);
```

When we call `countLetter`, we pass the value for the word inputted by the user (which gets stored in the `str` parameter variable) and the value for the character inputted by the user (which gets stored in the `letter` parameter variable). The method returns the number of times that letter appears in the string, and this result gets stored in the `numTimes` variable.

How Parameters Work

Up until this point, we have glossed over exactly what happens when we pass parameters to methods.

Suppose our full program looked like this:

```
using System;
public class TestParams {
     public static void main(String[] args) {
          int val = 4;
          TestParams.addOne(val);
          System.out.println(val);

          int[] array = {1, 2, 3};
          TestParams.zeroArray(array);
          for (int x : array) {
               System.out.println(x);
          }
     }

     public static void addOne(int num) {
          num++;
     }

     public static void zeroArray(int[] arr) {
          for (int i = 0; i < arr.length; i++) {
               arr[i] = 0;
          }
     }
}
```

It turns out that when we pass parameters that are ints, doubles, chars, or bools, then they are passed *by value*. This means that when we do something like:

```
TestParams.addOne(val);
```

Then the VALUE of `val` is passed – not `val` itself. This means that the value 4 is passed, and so the `num` parameter in the `addOne` method gets set to 4. We then add one to the `num` parameter, so that it has the value 5. When we finish executing this method, we return back to the `main` method after the call to `addOne`. At this point, we print out the value of `val`. **This**

print statement will print out 4. Even though we added one to `num` in `addOne`, `num` IS NOT `val` – it was just initialized to have the same value that `val` did.

On the other hand, arrays (and later *objects*) are passed *by reference*. This means that when we do something like this:

```
TestParams.zeroArray(array);
```

Then what we are really passing is the address of `array` in memory. Then, the `arr` parameter will be initialized to reference the same array in memory. When we set every element in `arr` (which is the SAME array as `array`) to zero, then it does change the original array. So when we print out `array` in `main` after the call to `zeroArray`, it will print out all 0's for the elements (because every element is set to zero in the `zeroArray` method).

From Main Method to Multiple Methods

Now that we've seen how to write separate methods, we need to learn how to reorganize our `main` method-only programs to use multiple methods. Here are some general rules for organizing your programs:

- If you want to repeat the same action more than once, put it in a separate method
- It's best when you can see an entire method on the screen without having to scroll. If a method is getting longer than that, consider splitting it up

Suppose we want to write a program that plays a simplified version of Hangman. We will ask the user for a word, and then print out a _ for each letter. We will then repeatedly have the user to guess a letter and then replace each _ with the letter (if it was a correct substitution based on the original word). We will continue this process until the entire word has been guessed.

Here is a `main` method-only solution to our game:

```
import java.util.*;

public class Hangman {
	public static void main(String[] args) {
		Scanner s = new Scanner(System.in);
		System.out.print("Enter a word: ");
		String word = s.nextLine();

		String result = "";
		for (int i = 0; i < word.length(); i++) {
			result += "_";
		}

		System.out.println("\nCurrent word: " + result);
		while (!(result.equals(word))) {
			System.out.print("\nGuess a letter: ");
			char letter = (s.nextLine()).charAt(0);

			boolean contains = false;
```

```
                for (int i = 0; i < word.length(); i++) {
                        if (word.charAt(i) == letter) {
                                contains = true;
                                result = result.substring(0, i) +
                                        letter + result.Substring(i+1);
                        }
                }

                if (contains == true) {
                        System.out.println("\nCurrent word: " + result);
                }
                else {
                        System.out.println("\nLetter is not in the word");
                }
        }

        System.out.println("\nYou guessed it!");
    }
}
```

Here are some separate methods we might consider writing:

- Building the intial string of “_____”
- Guessing a letter, and returning the updated string with that letter substituted in
- Printing the results after each step

Here is the updated program:

```
import java.util.*;

public class Hangman {
    //s can be seen throughout the file, by all methods
    public static Scanner s;

    public static void main(String[] args) {
        s = new Scanner(System.in);
        System.out.println("Enter a word: ");
        String word = s.nextLine();

        String result = Hangman.init(word.length());

        System.out.println("\nCurrent word: " + result);
        while (!(result.equals(word))) {
            String update = Hangman.guessLetter(word, result);

            //result!=update is true if a letter has changed,
            //and false otherwise
            boolean changed = !(result.equals(update));
            Hangman.printResults(update, changed);
            result = update;
        }

        System.out.println("\nYou guessed it!");
    }
```

```
	//returns a string of size '_' characters
	public static String init(int size) {
		String result = "";
		for (int i = 0; i < size; i++) {
			result += "_";
		}

		return result;
	}

	//asks the user for a letter
	//replaces all _ characters in cur when the corresponding
	//character in orig matches the input letter
	//return the updated orig string
	public static string guessLetter(String orig, String cur) {
		System.out.print("\nGuess a letter: ");
		char letter = (s.nextLine()).charAt(0);

		for (int i = 0; i < orig.length(); i++) {
			if (orig.charAt(i) == letter) {
				cur = cur.substring(0, i) +
					letter + cur.substring(i+1);
			}
		}

		return cur;
	}

	//if update is true, print cur
	//otherwise, print that the most recent letter wasn't in our word
	public static void printResults(String cur, boolean update) {
		if (update == true) {
			System.out.println("\nCurrent word: " + cur);
		}
		else {
			System.out.println("\nThe letter is not in the word");
		}
	}
}
```

Notice that we can still look at our `main` method to figure out the flow of the program (the order in which things happen). However, the `main` method is now a lot easier to read because the details have been passed off to other methods.