Methods/Scope

Ryan Nixon

Leaving the Main() Method

- So far all of the code that we have written has been inside of the main() method, the entry point for the program

- This doesn't have to be the case. Programmers can define additional methods to support more complex structures and to better organize the program

- Note that having multiple methods does not add anything to the language that isn't already there. All of the functionality could be gained by using repetition and numerous variables in the main() method...but that would be extremely hard to maintain

A Method

- `public static void main(String[] args)`

- The main() method that you have been using actually already contains all elements that your other methods will contain:

- `<privacy level> <class/instance level> <return type> <method name>(<parameter>[, <parameter>])`

- Note that the order matters for these elements. You should memorize this order; you'll be using it a lot

- We'll take a look at each one of these items today

Privacy Level

- The first item you specify for a method is its privacy level. This may be one of the following:
  - `public` - visible to all other classes in the program
  - `private` - visible only to the current class
  - `protected` - visible to the current class and its direct descendants. We'll cover this one a bit later
- These privacy levels allow a programmer to define functionality for a class that is specific to it and shouldn't ever be used outside of the class. This may include manipulating internal variables or helping another method out with repetitive logic

Class/Instance Level

- The second specifier for methods is the class/instance level. This alters how other classes interact with your method

- If you use `static` in your method then it exposes at the class level, so if you had an object Rabbit that class might have a static method called createRabbit():

```
public static Rabbit createRabbit() {
     return new Rabbit();
}
```

- This would be called by another class using the class' name to get a Rabbit object:

```
Rabbit newRabbit = Rabbit.createRabbit();
```

- Note that as each class must have a unique name in your program you will only be able to define one Rabbit.createRabbit() method, ever

Class/Instance Level

- If you omit the `static` specifier then your method will expose itself at the instance (object) level

- This level belongs to each individual object that you create.This means that the method would go along with all Rabbit objects that Rabbit.createRabbit() ever returns:

```
public void hop() {
     stepsTaken++;
}
```

- The Hop() method above is specific to the Rabbit that is hopping. Each Rabbit may hop slightly differently, so this method should be at the instance level. It would be called like this:

```
Rabbit newRabbit = Rabbit.createRabbit();
newRabbit.hop();
newRabbit.hop();
```

Class/Instance Level

- It is up to the programmer to decide which methods are static and which are not
- Your rule of thumb should be, “If the method applies to every object of the class make it static. If the method applies to *each individual object* make it non-static.”
- Java employs many more instance (non-static) methods than class (static) methods, but for today we’re sticking with class methods.

Class/Instance Levels

- But what about main()?
- Main is static because it is the entry point for your program. There should only be one entry point ever defined in a program, so main() will always be defined at the class level
- To enforce Java best practices you should name the class containing your method something that implies its importance
- This generally is the name of your program...or even “Program.java” would work.

Return Type

- The third specifier is the return type. This is a data type that your method will “return”, or provide to the calling method.

- You’ve seen this already with Scanner. The Scanner class has definitions that look like:

```
public String nextLine();
public double nextDouble();
public int nextInt();
```

- In each of these the return type allows the method to give data back to whatever is using it. Think of an equation in math, where you might have *a + b = c*. In this case your equation is “returning” the value of *c*.

- In Java you may only return one value, but this isn’t as bad as it sounds. You can return any object, and that object may have as many values in it as you like

Return Type

- You can also choose to return nothing at all. This is the `void` return type, and implies that your method performs something that doesn't need feedback. Main() is void because returning anything from it would fall on deaf ears.

- (this isn't actually the case -- if you return an integer from your main() then you can specify error codes for it at the command prompt, but we're sticking with void)

- Lets see a few of these return types in action

Return Type

- ```
  public int add(int a, int b) {
      return a + b;
  }
  ```

- ```
  public void increaseVolume() {
      volume = add(volume, 2);
      //Note the lack of a return
  }
  ```

- ```
  public int getVolume() {
      return volume;
  }
  ```
  ```

Method Name

- The next specifier is the name of the method. Nothing too special here
- Method names should conform to the same conventions that variables conform to.
  - Use camelCase capitalization
  - Don’t use any $pecial characters or spaces
  - All method names must be unique within their class scope

Method Name

- You should also use names for your methods that are helpful to programmers by implying what they actually do. play() and pause() methods for example might start playing media or pause it, respectively. If they did anything other than that it might confuse the people that are trying to call that method

- ```
  public void play() {
      state = PLAYING;
  }
  ```

- Note the use of a class constant here. Remember that class constants are defined outside of any methods and use `final`:

- ```
  public final int PLAYING = 3;
  ```

Parameters

- The final specifier, surrounded by parentheses after the method name, is for parameters. Parameters may be repeated as many times as you would like separated by commas

- Each parameter begins with its data type followed by its name. The name follows the same rules as a variable declaration

- `public int add(int a, int b)`

- The order of these parameters don't matter to your method, but *do* matter to the methods calling it:

- `int number = add(3, 5);`

- In the example above, `a` would be set to 3 and `b` would be set to 5.

Arguments

- I will often refer to parameters as arguments. These two terms are mostly interchangeable, but typically parameters apply to the method *definition* and arguments apply to the code that *calls* that method

- ```
  //A parameter
  public double convert(double param)
  ```

- ```
  convert(3.5); //An argument
  ```

Some Examples

The Stack

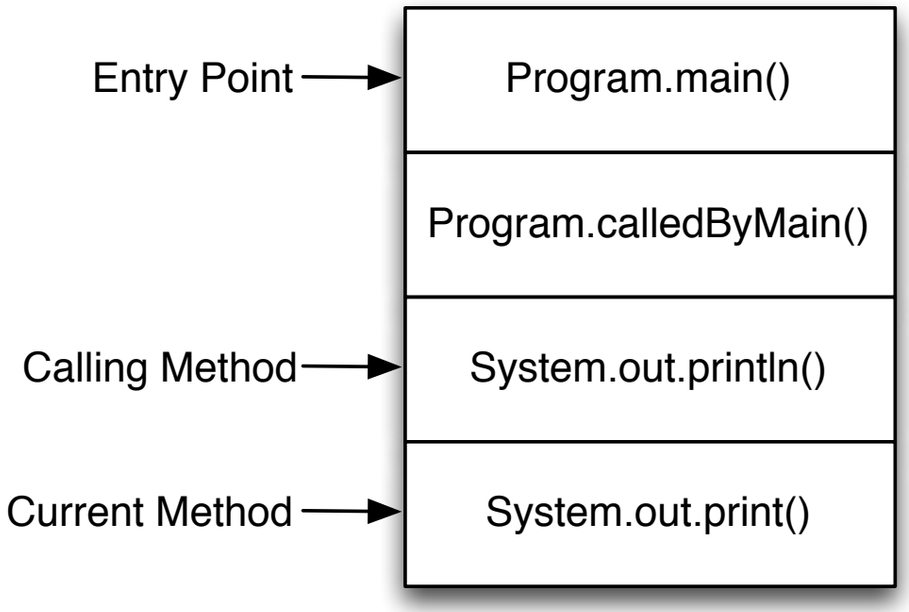
- In Java there is no real limit to the number of methods that can be called. Methods may call other methods, which in turn call there own methods

- Sometimes this can be confusing when it gets too deep. In these cases it helps to visualize the “stack”

- The stack is Java’s way of keeping track of which methods are calling which. You’ve seen this in the debugger with the “stack trace”

The Stack

- The stack in reality is an in-order list of all methods currently being called.
- When a method calls another method, the new method is added to the end of the stack
- When the current method returns, it is removed from the end of the stack
- In this way, the item at the end of the stack is always the currently running method



Data Hiding

- As method/variable names can be repeated at different scopes, sometimes it helps to explicitly specify which item you meant

- In Java, when you use a method or variable, it will look up the stack until it finds the nearest item by that name

Data Hiding

- ```
  public int aNumber = 5; //Hidden
  ```

- ```
  public void printNumber(int aNumber) {
      System.out.println(aNumber);
  }
  ```

- This example would print whatever is provided to the parameter...but what if we actually wanted the variable that was defined at the class level?

- There are two ways to do this, depending on whether the variable is static or non-static

Data Hiding

- **Static variables:**

```
class DataHiding {
     public static int classLevel = 5;

     public void poorlyNamed(int classLevel) {
          System.out.println(DataHiding.classLevel);
     }
}
```

- For static (class-level) variables, you may simply specify the name of the class, a period, then the name of the variable. This is exactly how you would use the variable (or methods for that matter) from another class

Data Hiding

- **Non-static variables:**

```
class DataHiding {
      public int classLevel = 5;

      public void poorlyNamed(int classLevel) {
          System.out.println(this.classLevel);
      }
}
```

- For non-static (instance-level) variables, you use the keyword `this`. The `this` keyword by name and function emphasizes, “the current object that I am” and overrides the logic of, “the current method that I am in”

- Using `this` allows you to call methods and use variables outside of your method scope. If you do not specify `this` and you use an instance-level variable or method, the `this` will be used implicitly

Another Example

Questions?