CMSC131

More Details about Methods

Structured Programing

One common philosophy/style/paradigm in computer programming is called “structured programming” where the program is designed to be composed of relatively small and logically self-contained blocks of code.

In an object-oriented programming language like Java, language elements such as classes and methods and objects provide a powerful set of tools that can help facilitate this style.

Classes, Methods, Objects

In our discussion of Java's **String** class we saw that it was an example of a **class** that described a structure to hold information (String **objects**) as well as have operations that can be performed on those objects (**instance methods**).

In our discussion of conditional statements and logic we saw how a **static method** could be added to a class (the **isOdd** method) to hold a block of code that might be used multiple times, but that was not itself associated with a specific object.

Adding Methods

A class can essentially have as many static or instance methods as you desire.

As we continue, we will explore the differences between these in more detail.

For now, while we will discuss both, we will only add static methods to our code examples.

Why use methods?

We will add methods either to:

- Simplify the readability of our code.
- Reduce redundancy in our code (eg: if you have a large block of code doing the same thing in two places, maybe you should create a method with that code and invoke the method from the two places).

The main Method

Every Java example we've seen so far has been a class with a meaningful name which contains a **static method** named **main** as the starting point.

```
public class MeaningfulName {
   public static void main(String[] args) {
    //code goes here…
  }
}
```

This is main method is a language convention to allow the JVM to know where a program should begin. This method can call other static methods, create objects, and use objects to invoke instance methods.

Objects

We have seen a few examples where we instantiate an object of a particular class and access members of those classes, specifically with the **`Scanner`** class and the **`String`** class.

```
Scanner sc = new Scanner(System.in);
int x = sc.nextInt();
...
String answerHolder = sc.next();
answerHolder = answerHolder.toLowerCase();
```

The variables **`sc`** and **`answerHolder`** are references to distinct objects with access to instance methods within their classes.

Instance

When a method or field is not marked as **static**, it is an **instance** one instead (no keyword used to denote that).

Instance methods have one copy of their code in existence, but when executed they always have a specific object associated with them; these methods are always invoked via an object, and that object is the one associated with that execution of the code.

Instance fields are created in memory every time an object is created, and are logically connected with that object. Once the object is removed from memory, all its instance fields are too.

String Objects

Within the **`String`** class, we are provided a number of useful *instance* methods.

For example, **.charAt(int)** and **.length()**

```
Scanner sc = new Scanner(System.in);
String word = sc.next();
System.out.println("The length is " + word.length());
System.out.println("One character in it is " + word.charAt(2));
```

A reminder and a note on the above:

- The “0th” character would be the one we might (as humans) think of as first…
- If the user entered a one or two character word, this would crash! Why?

What letter is printed if the user enters **abcde** ?

17% A. a

17% B. b

17% C. c

17% D. d

17% E. e

17% F. The program crashes…

```
String word = sc.next();
System.out.println(word.charAt(2));
```

Response Counter

Reminder: static vs instance methods

Static methods are associated with the ***class as a whole***, not specific object instances. We will be discussing these more here.

Non-static methods (which we call instance methods) are associated with a ***specific object*** and can act upon things stored in that one object when invoked. While we are using these, we are not creating our own yet.

Static Method Prototype

```
public static return_type method_name (parameter_list) {
    body_of_method
}
```

All of our `static` method definitions will follow the above syntax.

- You choose the appropriate return type.
- You choose the meaningful method name.
- You choose what information needs to be passed into the method.

  Note: If something isn't passed into the method, that method doesn't know about it even if another method, even the `main` method, does.

Reminder: static method

```
public class SimpleDoWhileWithMethod {
  public static void main(String[] args) {
    int userValue;
    Scanner sc = new Scanner(System.in);
    do {
      System.out.print(
          "Enter an odd number to continue: "
      );
      userValue = sc.nextInt();
    } while (!isOdd(userValue));
    System.out.println("Thank you.");
  }

  public static boolean isOdd (int num) {
    return (num%2)!=0;
  }
}
```

Memory Model

Memory is organized into three logical regions in Java; the stack, the heap, and metaspace .

We will start to discuss these more now, since understanding where things “live” in memory can help understand why certain things work the way they do with methods and with objects.

Objects, instance methods, memory…

Consider the following code segment:

```
String firstName = "Evan";
String lastName = "Golub";
…
firstName = firstName.toLowerCase();
lastName = lastName.toUpperCase();
…
```

What do firstName and lastName now contain if we were to print them?   Let's draw this in memory…

DoSomeMath.java example

```
public class DoSomeMath {
  public static void main(String[] args) {
  int x;
  int y;
  int z;

     x = 17;
     y = 23;
     z = 14;

     printSum(x,y);
     printSum(y,z);
  }

  public static void printSum(int first, int second){
     System.out.println(first+second);
  }
}
```

Variables

In the DoSomeMath example, there were some local variables…

declared

```
 int x;
 int y;
```

assigned to

```
 x = 17;
 y = 23;
```

and used

```
 printSum(x,y);
```

In short, variables have a data type (such as int) and refer to space within the computer's memory where their values (such as 17) are stored.

You can assign values to them and read those values back out from them.

At-home exercise: trace DoSomeMoreMath.java

```
public class DoSomeMoreMath {
  public static void main(String[] args) {
  int x, y;
     x = 17;
     y = 23;
     printSum(x,y);
     printProd(x,y);
     printQuot(x,y);
     printQuot(y,x);
  }
  public static void printSum(int first, int second){
     System.out.println(first+second);
  }
  public static void printProd(int alpha, int beta){
     System.out.println(alpha*beta);
  }
  public static void printQuot(int alpha, int second){
     System.out.println(alpha/second);
  }
}
```

AddOne.java example

```
public class AddOne {
  public static void main(String[] args) {
  int x;
     x = 17;
     System.out.println(x);
     printPlus1(x);
     System.out.println(x);
  }
  public static void printPlus1(int val){
     val = val + 1;
     System.out.println(val);
  }
}
```

Draw the memory model now to trace this.
Have an answer ready…

What will that print?

- 20% A. 17, 17, 17
- 20% B. 17, 18, 17
- 20% C. 17, 18, 18
- 20% D. 17, 18, 19
- 20% E. Something else.

Response Counter

When calling methods…

Two key things to always consider:

– If the method returns a value, then the statement that calls it should deal with that value somehow (use it, store it, etc).

– The calling statement's argument list needs to match up with the method's parameter list by data type.

- In some situations, Java can detect a mismatch and automatically convert the argument so that its data type matches that defined by the method as the required parameter type.

Static Fields

Similar to how a class can have static methods that are not associated with any particular object, a class can also have pieces of information (fields) that are not associated with any particular object but rather with the class as a whole.

– This is useful when you have some information related to the class as a whole; different objects do not have different values. This is sometimes referred to as shared information.

One thing that is important to pay attention to is your naming convention. If a local variable within a method has the same name as a field of the class, the local variable will "hide" that field within the scope of that method.

Exploration Program: What will it print?

```
public class ExploreStaticVariables {
  static int howManyOdd=0, howManyEven=0;

  public static void main(String[] args) {
    for (int count=0; count<10; count++) {
       System.out.println(count + " " + printParity(count));
    }
    System.out.print("I saw " + howManyOdd + " odd.");
    System.out.println("  I saw " + howManyEven + " even.");
    //What do you think will be printed here?
    //Draw the memory model to trace this...  Have an answer ready...
  }

  public static String printParity (int num) {
    String retStr="";
    int howManyEven=0;
    if ( (num%2)!=0 ) {howManyOdd++; retStr+="is odd.";}
    else {howManyEven++; retStr+="is even.";}
    return retStr;
  }
}
```

At the end, what will it print?

20% A. I saw 0 odd. I saw 0 even.

20% B. I saw 5 odd. I saw 0 even.

20% C. I saw 0 odd. I saw 5 even.

20% D. I saw 5 odd. I saw 5 even.

20% E. Something else.

Response Counter

Static Overview

When a method or field is marked as **static** there is a single one of them that is available to be used.

- If multiple parts of the program have access to them, then all parts of the program share that single one.
- Once created they continue to exist until the program terminates.

Static methods have no specific object associated with them.

Static fields are created in memory the first time that the JVM loads their class and are never removed from memory.