CSC 325 Adv Data Structures

Lecture 5

The Static Keyword

Introduction

There are concepts I've noticed students struggling with all the way to senior year. I'm hoping to fix these issues before it gets to that point. To that end, I will go over the static keyword, the final keyword and generics (all in separate lessons).
The "static" keyword means different (but similar) things based on its usage. After this lesson, if you still don't understand, then you need to re-read these notes until you do. When this class is over, none of you should be able to claim you don't understand these concepts.

Java's Static Keyword on Methods and Fields

The "static" keyword can be placed on a method, a field, or a class. In fact, it is used by many different languages and means something slightly different in each language and in each usage.

In Java, we can put static on a method or a field. This turns that method/field into a class method/field instead of an instance method/field. The only practical difference is whether we require an instance or not.

```
class ExampleClass {
  public int num1 = 8;
  public static int num2 = 5;
}

class MainClass {
  public static void main(String[] args) {
    ExampleClass instance = new ExampleClass();
    System.out.println(instance.num1);
    System.out.println(ExampleClass.num2);
  }
}
```

Output:

```
$ javac Example.java
$ java MainClass
8
5
```

Essentially, an instance variable is tied to an instance and a class variable is tied to the class. This is the same for methods:

```
class ExampleMethods {
  public int foo() {
    return 3;
  }

  public static int bar() {
    return 5;
  }
}

class MainClass {
  public static void main(String[] args) {
    ExampleMethods instance = new ExampleMethods();
    System.out.println(instance.foo());
    System.out.println(ExampleMethods.bar());
  }
}
```

Output:

```
$ javac ExampleMethods.java
$ java MainClass
3
5
```

As you can see, the same rules apply for methods. An instance method is tied to an instance and a class method is tied to the class.

Now, to understand this next bit, you will have to understand how the “this” pointer works. We know that if we have an instance method, we can use the “this” keyword to point to the current instance. For example:

```
class SomeClass {
  private int var = 3;

  public int getVar() {
    return this.var;
  }

  public void changeVar(int var) {
    this.var = var;
  }
}
```

But what exactly is “this”? And how is it declared. Well, the Java compiler actually sees the functions above, like this (after it compiles the code):

```
class SomeClass {
  private int var = 3;

  public static int getVar(SomeClass this) {
    return this.var;
  }

  public static void changeVar(SomeClass this, int var) {
    this.var = var;
  }
}
```

A reference to the class gets put into the parameter list, and the reference name is “this”. This would mean that “this” is just an object of the class. When the function is called, we normally write something like this:

```
class Main {
  public static void main(String[] args) {
    SomeClass sc = new SomeClass();
    sc.getVar();
    sc.changeVar(42);
  }
}
```

Actually, the compiler sees it more like this:

```
class Main {
  public static void main(String[] args) {
    SomeClass sc = new SomeClass();
    SomeClass.getVar(sc);
    SomeClass.changeVar(sc, 42);
  }
}
```

So you see, the instance automatically gets placed as the first parameter in the method and the instance automatically gets put as the first argument when calling the method. You never see this, but this is how it works[1]!

This means that the only difference between a static and a non-static method is the inclusion of a “this” reference. Basically, after the compiler is done, instance (aka non-static) methods are given a “this” parameter as the first parameter and class (aka static) methods are not.

[1] Actually, the REAL process involves saving the fields into a structure and name mangling the methods (best ref I could find: https://9fans.github.io/usr/local/plan9/src/libmach/gxxint_15.html)

Let's combine this with what we already know. We already know that an instance variable requires an instance. Since instance methods are given the "this" pointer and since that is, in fact, an instance, this means that instance methods can refer to instance variables.

```
class A {
  private int var = 3;

  public int foo() {
    return this.var;
  }
}
```

"var" is an instance variable and "this" is an instance, hence "this.var" is the instance variable "var" being referenced using the instance "this". Since this methods is not static, the Java compiler will add "this" as the first parameter to the method, so we are given an instance to access the instance variables on (as well as the instance methods on).

However, since static methods are not given an instance, we cannot access instance variables. So a static method can only access static fields.

```
class A {
  private int var = 3;
  private static var2 = 5;

  public int foo() {
    return this.var;
  }
  public static int bar() {
    return A.var2;
  }
}
```

Again, if we remember "an instance variable is tied to an instance and a class variable is tied to the class" we can see why the code above works. For an instance method, we are given the "this" instance. Therefore we can refer to the instance variables on that instance. For a class method (which is our static method), we are NOT given an instance and can therefore only access class variables (since we always have access to the class itself). Now, hopefully the following won't confuse you but, just because we have access to an instance (i.e. "this"), doesn't mean we have to use only that. We can always refer to the class variables as well:

```
class A {
  private int var = 3;
  private static var2 = 5;

  public int foo() {
    return this.var + A.var2;
  }
  public static int bar() {
    return A.var2;
```

```
  }
}
```

An instance method can refer to any instance fields (or methods) but it can also refer to class fields (or methods) since those are ALWAYS available (given that their visibility (i.e. public or private) is adhered to). However, a class method doesn't have an instance. So a class method can only refer to class fields and class methods. That is, unless we create an instance:

```
class A {
  private int var = 3;
  private static int var2 = 5;

  public int foo() {
    return this.var + A.var2;
  }
  public static int bar() {
    A a = new A();
    return a.var;
  }
}
```

In bar we create a new instance and refer to that instance's fields. This isn't a workaround, however, as we still don't have access to the current instance, instead we are creating a brand new instance and using that. Any other instance of this class other than the one we just created is unaffected.

There is even a concept of a "static block". This special function is used kind of like a constructor but instead of being called when an object is instantiated, it is called when a static member is referenced.

```
class A {
  public int var = 3;
  public static int var2;

  static {
    var2 = 5;
  }
}

class Main {
  public static void main(String[] args) {
    System.out.println(A.var2);
  }
}
```

The static block is automatically called when A.var2 is referenced in main. As soon as any static member is referred, the static block is called, but only for the very first time. Every time a static member is referred to after that, the static block isn't executed.

This makes sense and in effect, you can think of the static block as a constructor for the static fields. Just like a normal constructor allows you to initialize instance fields, a static block allows you to initialize class fields:

```
class A {
  public int var;
  public static int var2;

  static {
    var2 = 5;
  }

  public A() {
    this.var = 3;
  }
}

class Main {
  public static void main(String[] args) {
    A a = new A();
    System.out.println(a.var);
    System.out.println(A.var2);
  }
}
```

Java's Static Keyword on Classes

Consider the following Java code:

```
class LinkedList {
  public static class Node {
    public int data;
    public Node next;

    public Node(int data) {
      this.data = data;
    }
  }

  private Node head;

  public LinkedList() {
    head = null;
```

```
  }
  public void addNode(Node node) { /* implementation not shown */ }
}
public class StaticClassTest {
  public static void main(String[] args) {
    LinkedList list = new LinkedList();
    LinkedList.Node node = new LinkedList.Node(5);
    list.addNode(node);
  }
}
```

You can actually nest classes in Java. In this case, the Node class is nested within the LinkedList class. This gives the LinkedList class access to the Node class just as it would if Node were not nested (e.g. it can instantiate objects of Node like normal), but the access outside of the LinkedList class is a little different. Since Node is part of LinkedList, we must specify LinkedList.Node to refer to this class. So in our main method, the "node" variable we are creating is of type "LinkedList.Node". But how do we instantiate this node? Well, if we were referring to a static field, we could do the following:

```
LinkedList.nameOfStaticField
```

If we were referring to a static method, we could do the following:

```
LinkedList.nameOfStaticMethod()
```

Since we are referring to a static class, we do the following:

```
LinkedList.NameOfStaticClass
```

And so when instantiating this object, we write:

```
LinkedList.Node node = new LinkedList.Node(5);
```

Makes sense and is consistent with how the static keyword is used. Things get a little weird though if we don't have the Node class marked as static. Consider what would happen if the above code was changed to:

```
class LinkedList {
  public class Node {
    public int data;
    public Node next;

    public Node(int data) {
      this.data = data;
    }
  }

  private Node head;

  public LinkedList() {
    head = null;
  }
```

```
  public void addNode(Node node) { /* implementation not shown */ }
}
```

In this case, how do we instantiate a new node? Well, if we were referring to a non-static field, we could write:

```
list.nameOfNonStaticField
```

If we were referring to a non-static method, we could write:

```
list.nameOfNonStaticMethod()
```

Since we are referring to a non-static class, we write;

```
list.NameOfNonStaticClass
```

and if we want to instantiate an object of this class, we will need to write:

```
list.new NameOfNonStaticClass()
```

And so our main function becomes:

```
public class StaticClassTest {
  public static void main(String[] args) {
    LinkedList list = new LinkedList();
    LinkedList.Node node1 = list.new Node(5);
    LinkedList.Node node2 = new LinkedList().new Node(5);
    list.addNode(node1);
    list.addNode(node2);
  }
}
```

Both node1 and node2 show valid ways of creating a new Node object. Remember, if you have an instance (aka non-static) member, you need an instance to refer to it.

Static Keyword in Other Languages

C and C++ uses the static keyword on functions and local variables. If you put static on a function, the function is only available for use in that file. Otherwise, the function can be used in other files.

Filename: static_func.c	**Filename: main.c**

`static int foo() {`
`  int counter = 0;`
`  counter++;`
`  return counter;`
`}`
	`#include <stdio.h>`
	
	`extern int foo(void);`
	
	`int main() {`
	`  printf("%d\n", foo());`
	`}`

If you try to compile both files above, you will get this:

```
$ gcc static_func.c main.c
c:/mingw/bin/../lib/gcc/mingw32/9.2.0/../../../../mingw32/bin/ld
.exe: main.c:(.text+0xf): undefined reference to `foo'
collect2.exe: error: ld returned 1 exit status
```

Basically, the compiler won't allow the foo function to be used in main.c since it was declared as static in static_func.c. If you remove this static keyword, the code above works just fine when compiled and will output 1 when executed:

```
$ gcc static_func.c main.c
$ ./a.exe
1
```

You can also put static on local variables. If you do, the variable's value will be remembered from the last time the function was called:

```c
#include <stdio.h>

int foo() {
  static int counter = 0;
  counter++;
  return counter;
}

int main() {
  printf("%d\n", foo());
  printf("%d\n", foo());
  printf("%d\n", foo());
}
```

This C code will produce the following output:

```
1
2
3
```

Because the variable "counter" was marked as static, it retained its value between function calls and therefore each time the function was called, it was not re-initialized and instead just incremented by one.

If you want the function to be restricted to just that file AND remember the values of local variables inside of the function, you would use static on both:

```c
static int foo() {
  static int counter = 0;
  counter++;
  return counter;
```

```
}
```

In fact, you could have multiple uses of the static keyword:

Filename: static_func.c

```
static float baz() {
  float number = 1.0f;
  return number;
}

static int foo() {
  static int counter = 0;
  counter += baz();
  return counter;
}

char bar() {
  static char letter = 'z';
  letter -= foo();
  return letter;
}
```

Filename: main.c

```
#include <stdio.h>

extern char bar(void);

int main() {
  printf("%c\n", bar());
  printf("%c\n", bar());
  printf("%c\n", bar());
}
```

Output:

```
$ gcc static_func.c main.c && ./a.exe
y
w
t
```

In C#, the word static can be used on a field of a class. If it is, it follows the same rules as Java. The field is bound to the class and becomes a “class variable”, meaning all instances of that class share the same value. When it changes for one, it changes for all. However, if static is used on a class, it means that all members of that class (all fields and methods) must be static. Note that this definition of a “static class” is NOT the same in Java, as Java static classes are completely different. Example of a static class in C#:

Filename: static.cs

```
using System;

public static class StaticClass {
  private static int   num1 = 1;
  public  static float num2 = 3.14f;

  public static int Foo() {
    return num1 * 5;
  }

  public static void PrintSomething() {
    Console.WriteLine("Something");
  }
}

public class Program {
  public static void Main(string[] args) {
    Console.WriteLine(StaticClass.Foo());
    StaticClass.PrintSomething();
    Console.WriteLine(StaticClass.num2);
  }
}
```

Output:

```
$ csc static.cs
Microsoft (R) Visual C# Compiler version 3.11.0-4.22108.8
(d9bef045)
Copyright (C) Microsoft Corporation. All rights reserved.

$ ./static.exe
5
Something
3.14
```

In fact, if we omit the static keyword on any member of a static class, we get an error:

```
using System;

public static class StaticClass {
  private int    num1 = 1;
  public  static float num2 = 3.14f;

  public static int Foo() {
    return num1 * 5;
  }

  public static void PrintSomething() {
    Console.WriteLine("Something");
  }
}

public class Program {
  public static void Main(string[] args) {
    Console.WriteLine(StaticClass.Foo());
    StaticClass.PrintSomething();
    Console.WriteLine(StaticClass.num2);
  }
}
```

Output:

```
$ csc static.cs
Microsoft (R) Visual C# Compiler version 3.11.0-4.22108.8
(d9bef045)
Copyright (C) Microsoft Corporation. All rights reserved.

static.cs(4,17): error CS0708: 'StaticClass.num1': cannot
declare instance members in a static class
```

As you can see, we cannot declare instance members (such as instance variables and instance methods) inside of a static class. We can only declare static members.