Hello! Today we will focus on the static keyword, calling conventions of methods, and what scope and lifetime mean in Java.

Now that our chapters will tend to generate multiple files, I strongly suggest you create a folder for each chapter, and then just add files to that folder.

Naming Things. Quick note on names: when we create a variable in a class – whether we make it static or not – we can refer to it as a field or attribute. We may refer to both attributes and methods as members of the class. Although we have strived to use the same names for things, these words are commonly used to refer to the variables and methods in a class.

Methods – Static Methods

The way in Java to name a block of code, and then be able to call it elsewhere, is to write a method. A method always goes in a class definition.

We have one major distinction to make, though: should the method be static or not? A static method can be called all by itself (no object of the enclosing class needed), much like a function in Python. A non-static method in Java is written in a class and associated with a particular object and thus can use the object’s instance variables. Being non-static is the default behavior in Java, specified by the absence of the static modifier. We focus first in this tutorial on writing static methods.

Static Method Example
Contrast a normal method this with the following doTheTask method:

```java
public class Test {
    public static void main (String[] args) {
        int x = 10;
        String s = "hello";
        int res = doTheTask(x,s);
        System.out.println("result: " + res);
    }

    public static int doTheTask(int times, String thing) {
        int max = 8; // don't print too often
        int i; // declared here so we can use its value after the loop
        for (i=0; i<times && i <= max; i++) {
```
```java
System.out.println(thing);
}
return i;
}
```

We actually have two static methods here: `main`, and `doTheTask`. Whenever we run a program, there's no action that creates a `Test` object – the main method is run directly, as if we had called `Test.main(args)` from somewhere else. Indeed, we can do exactly that:

```java
public class OtherTest {
    public static void main(String[] args){
        Test.main();
        Test.doTheTask(5,"hi.");
    }
}
```

Quick understanding question: how many times did `doTheTask` get called due to running the main method above?

→ Twice: once because `Test.main` calls it, and once because `OtherTest`'s main method calls it directly.

**Your Turn!**

1. Make a class named `MyMathStuff`, and then add these static methods to it:
   a. `sumArray`, which accepts an array of integers and returns the sum of the array
   b. `maxIndex`, which accepts an array of integers and returns the index of the maximum value in the array. (When no max exists for a length-zero array, return -1).
2. Create another class, `TestMyMathStuff`, which does not create a `MyMathStuff` object but still uses each method and prints out the results.
3. Add a static method to `TestMyMathStuff`, named `useMMS`, which creates an array, calls `sumArray` and `maxIndex`, and then prints each. Call it from `TestMyMathStuff`'s `main` method.

**Static Variables**

Variables can be static, too. Instead of calling them "instance variables" as we did for non-static variables (to indicate that we have one variable per object, or instance), we call these static variables **class members** to indicate that we have one shared value for the entire class. It will always exist (after the first time the class is referenced). The most direct examples of this are also usually constants: `Math.PI`, `Integer.MAX_VALUE`, and `Integer.MIN_VALUE`. Because they are declared static, there is only one copy stored on the computer and we access that value via the class name. As a separate choice, making them final means that they cannot be changed – it is just the most meaningful to store the value of pi in the `Math` class, and the maximum and minimum values that an `int` can store in the `Integer` class.

When we create a static variable, since we will not instantiate it per-object at constructor-call time, we do actually initialize static variables at their declaration. (If we don't, a default value of `0/false'/''/null is supplied).
Just a quick note: our declarations are still following the pattern of `modifiers type identifier`. We are also just performing an instantiation at the same time. Keep in mind that `public`, `static`, and `final` are all modifiers. The above two lines of code are spaced out a bit oddly to show the grouping of the modifiers, type, and identifier.

To access a class member, we again just type `ClassName.memberName` (where we use the actual class definition's name, and the specific variable or method's name).

**Your Turn!**

4. Print out the value of pi using the static definition in the `Math` class.
5. Add a static variable to your `MyMathStuff` class to define your own silly version of pi as 3.0. Use it to print out the area of a circle with radius=5. (You don’t need to create and use a `Circle` class for this one usage, just do the calculation directly). How do you access your version of pi?
6. Longer Example: make a small class like `Square`, `Coordinate`, or `Sphere`.
   a. Add a static variable named `numCreated`, and initialize it to zero.
   b. In the constructor, increment this variable. You can also use the current value of `numCreated` during a constructor call to give a unique identification number to each object, sort of like a serial number on a musical instrument or the VIN on a vehicle.
   c. Note that in this way, the constructor caller doesn't have to track how many objects have been created – the class takes care of it. This effect is not achievable without static variables (or some piece of state outside of the class).

**Quick Notes on `static`**

- Assuming it is public, a static variable can be accessed by static and non-static methods within the same class. It can also be accessed anywhere the class containing it is accessible.
- A class can contain both static and non-static variables, and static and non-static methods.
- You can actually use an object of a class to access a static thing in that class. You would type `objectName.staticMember` to access it; this is unusual, and we’d prefer to see `ClassName.staticMember` instead.
- When we focus on visibility below, consider what it would mean to have a private static variable, or a private static method. Visibility and static-ness are two separate choices that we make for each member in a class.

**References, Scope, and Visibility**

Like in Python, primitive types in Java are stored directly, but reference types are stored indirectly, on the heap. This becomes relevant when passing arguments to a function. However, the rules in Java are the same as for Python: always copy what’s in the box, and always store a memory address for a non-primitive type, instead of the value itself.

**Your Turn!**

7. Trace through the following code in memory:
public class MemoryTrace {

    public static void main(String[] args){
        int[] numbers = {5, 7, 6, 3, -1, 2};
        int item = 6;
        int index = getIndex(item, numbers);
        if (index == -1)
            System.out.println("Item not found!");
        else
            System.out.println("Item found at index " + index);
        int[] arr = {6, 5, 2};
        System.out.println(getIndex(5, arr));
        //System.out.println("Can't print thing " + thing);
    }

    public static int getIndex(int item, int[] thing){
        int ctr = 0;
        boolean found = false;
        thing[1] = 0;
        while (ctr < thing.length && !found){
            if (thing[ctr] == item)
                found = true;
            else
                ctr++;
        }
        if (!found)
            ctr = -1;
        item = 999;
        int[] numbers = new int[6];
        return ctr;
    }
}

We need to be aware of when values and variables are accessible or not, and when they are created and destroyed. We say the scope of a variable is where in the source code we can refer to the variable, and we think of the lifetime of a value as the time from creation to destruction.

**Scope**

Scope refers to the region of a program in which a definition is available. Each variable, method, and even class definition has a scope. We first focus on the location of the definition:

**Locations and Scope:** Where something is defined, and where it is accessible.

- **local variable:** defined within a block of statements, such as inside a for-loop, inside a method, or other control structure. Accessible from declaration to end of the statement block.
- **parameter:** variable defined in a method signature (the method definition). Accessible through the entire body of its method. (Parameters are truly just local variables).
• **field: (instance variable or class variable):** defined in a class. Accessible through the entire class definition, except that static methods cannot access non-static fields.

• **class:** defined in a class. Accessible through the entire class definition.

Note: if we have a local variable that is currently referring to an object, and that object has (publicly visible) fields that we can also access, that doesn’t mean the interior thing we accessed is in scope here: we used the dot operator to go to a different scope and find something that was in scope over there.

For each of these, we should consider the definition’s lifetime: when the definition begins its existence, and when the definition ceases to exist.

A collective example is below.

→ The **scopes** of things are written on the left side

→ destruction points (deaths) are indicated on the right.

```
public class Bar {
  public int a;
  private int b;
  public static int c;

  public int doStuff (int x, int y) {
    int count = 0;
    for (int i=0; i<x; i++) {
      int temp = i;
      temp += y;
      if (temp%17==0) { return count; }
      if (temp%2==0) {
        String s = "even";
        System.out.println("looking "+s);
        count++;
      }
    }
    else { count--; }
    return count;
  }

  public void other () {
    return;
  }
}
```

• a, b, c, doStuff, other: live as long as the object containing them exists (has a reference to it).

• c lives the entire duration of the program (static fields are always there).

• doStuff, other are methods – don’t ‘die’, but only in scope with an object (they are non-static).

What do we do with scope and lifetimes?
So what do we do with knowledge of the scope and lifetime of things? Especially for variables, it helps us understand how we can use them.

- As long as an object has a living reference to it, Java will keep it around. Once no such references exist, Java marks it as garbage and effectively deletes it (specifically, Java reclaims that memory to use for storing other data).

We can look for and understand bugs, such as declaring a variable in too narrow a block of scope. As an example:

```java
for(int i=0; i<xs.length; i++) {
    int sum = 0;
    sum += xs[i];
}
System.out.println("sum = " + sum);
```

- Not only will it fail to compile (sum isn’t in scope when we try to print it), but sum lives and dies during each iteration of the loop.

- We can differentiate between separate variables that happen to have the same name. A local variable in one method has no connection to a parameter in another method, even if we happen to use the local variable’s value as the actual parameter value when calling the other method. We often name things the same because they represent the same sort of value at different times, and yet we forget to ensure the actual value is passed from one to the other.

### Your Turn!

8. The following code has a syntax error in it due to scope. Identify it (and correct the code, without removing any variables).

```java
public class Suspicious {
    public static int sum = 9000;

    public int max (int[] xs) {
        int sum = 0;
        for (int x = 0; x < xs.length; x++) {
            int temp = 0;
            temp = sum;
        }
        System.out.println("last sum seen: " + temp);
        return sum;
    }
}
```
Visibility

A large part of the OO philosophy revolves around encapsulation and abstraction. We want to be able to only interact with the exposed interface of a class, to such an extent that the language itself gives the programmer the ability to enforce hiding of the interior representations and other algorithmic decisions.

We have four possible visibilities: public, private, protected, and package-private.

- **public**: allow anyone with a reference to access this member.
- **private**: disallow access from outside the object. (An object can always access its fields and methods; its methods can always access its own members of any visibility).

So far, we have perhaps written all of our classes, methods, and instance variables as public. But now that we are more comfortable with classes and objects and the components that we use to create classes, we should focus more on enforcing the abstractions of OO-style programming by hiding members of our classes.

---

**Example**

If I have everything defined as public, and I am representing my house (and everything in it), then anyone who has my address can go inside and steal my sandwich. This is not good.

```
meanie.eat(myHouse . kitchen . fridge . sandwich); //sandwichless sadness.
```

But what if myHouse is locked? I can do this by making everything in my house private.

→ Now you can't have my sandwich. Java will protect it for me.

```
//sandwich haven via compilation error.
meanie.eat(myHouse . kitchen . fridge . sandwich);
```

→ **compilation error**: we can't access the kitchen when we're locked outside.

what if I have a public requestSandwich method? Then you can request my sandwich and I'll give it to you.

```
politePerson .eat (myHouse . requestSandwich ());
```

---

**Reading and Writing Privileges**

Making everything public is problematic. No matter how carefully we construct a class, and write wonderful methods that can be used to beautifully solve the task at hand, any user of our class is free to go in and directly modify any state, and call any method even when it's not appropriate to do so.

```
public fields offer unlimited read and write privileges.
```
Consider an example of a `BankAccount` class:

```java
class BankAccount {
    public int balance;
    ...
}
```

//elsewhere, outside the BankAccount class:
BankAccount ba = new Account();
ba.balance = 2100000000;  //bad for business!
exsAccount.balance = 0;   // bad for the customer!

Rather than allow anyone to access a field member, we can make it private so that both reading and writing privileges are revoked:

```java
class BankAccount {
    private int balance;
    ...
}
```

```java
ba.balance = 2100000000;  // compilation error
int secret = exsAccount.balance; // compilation error
```

How can we selectively restore reading and writing privileges? Through public methods that are in the same class as the private field. A field is always visible inside the class, so we can choose to return a copy of a private field from a public method in that same class.

- By creating a public method that happens to return a copy of the field, we can restore reading privileges. These methods are often called "accessors" or "getters".
- By creating a public method that happens to accept a parameter and update a field, we can restore writing privileges. These methods are often called "mutators" or "setters".
- For a field named `fooBar`, it is conventional to name the getter `getFooBar`, and the setter `setFooBar`.
- There is nothing special about a getter or setter method – Java treats them just like any other public method.

```java
public class Square {
    //private: revoke reading and writing privileges.
    private int side;

    public Square (int side) {
        this.side = side;
    }

    //restore reading privilege to side.
    public int getSide ( ) {
        return side;
    }

    //restore writing privilege to side. Perform extra checks!
    public void setSide(int side) {
        if (side > 0) {
            this.side = side;
        } else {
            this.side = 1;
        }
    }
}
```
**Good Practice:** The purist's view is that every single field should always be made private, and that we then choose whether reading/writing is allowed selectively by adding getter/setter methods. The only exception to this would be constants – since they can't be written anyways, it is safe to restore just reading privileges by making a public final field member.

→ if you want to make something final and public, this is a good compromise of usability and good encapsulation.

---

**Your Turn!**

9. **Rectangle Example**
   a. Create a Rectangle class, with fields for length and width. Make them both public initially.
   b. Make a public constructor.
   c. Create or reuse a testing class to create a Rectangle object, then modify its fields, and then print them out.
   d. Now, realize we want to write good OOP code, and change both fields to be private.
   e. Add getter and setter methods for both fields. Disallow non-positive values for each field.
   f. Go back to your testing class and edit the code to use the getters and setters.
   g. **Going Further:** Let's remove the setter for width (revoke writing privileges for width).
   h. Add public methods to get and set the perimeter. But don't add a perimeter field!
      a. In the setter, re-work the equation $2*(\text{length}+\text{width})=\text{perimeter}$ to figure out how to modify just the width to achieve the given perimeter.
      b. In the perimeter getter, calculate the perimeter based on the length and width.
         → notice how our internal representation can be substantially different from what the user sees? It appears that we have a perimeter field, but we have chosen a different implementation path. If we later change our minds, we can get rid of the width field and add a perimeter field instead, and not have to change the signatures of any of the methods in order to provide the same functionality.
   i. So let's try that: get rid of the width field, add a perimeter field, but don't change any methods’ signatures (the visibility/return type/name/parameters portion of the definition).

10. **BankAccount Example**
    a. Create a BankAccount class. Give it private fields for accountNum, balance (in pennies), password, and customerName.
    b. What getters and setters do you want to provide?
    c. Add a constructor, with parameters for each field.
    d. Add another constructor, with parameters for each field except balance, and set it to zero. This is an example of an overloaded method.
    e. Add a toString method that returns a nice String representation of the account. Do you want to include all fields in this printout? (Honest question – think of where you want this method to be used, and realize if it is public you have already decided where it can be used).
    f. Add withdraw, deposit, and changePassword methods. What should their visibility be? Should a copy of the password be a formal parameter in any of these?
    g. Add sufficient checks so that a user can't deposit a negative amount, withdraw a negative amount, or withdraw more than they have in the account.
    h. Are there any other checks you want to add?
    i. Test out your code.