Chapter 4
Methods

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 lab, 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**. We may refer to both fields 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.

Non-static Method Example

Here is an example of a non-static method, in the Square class: look at the perimeter method below.

```
public class Square {
	public int side;
	public Square (int side) {
		this.side = side;
	}
public int perimeter () {
		return (side * 4);
	}
}
```

In order to call the perimeter method, we **must** have a `Square` object, such as `sq.perimeter()` in the following code:

```
Square sq = new Square(5);
System.out.println("square's perimeter: " + sq.perimeter()  );
```

Static Method Example

Contrast this with the following doTheTask method:

```
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 a, String t) {
		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<a; i++) {
			System.out.println(t);
			if (i>max) {
				break;
			}
		}
		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:

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

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!

- Make a class named MyMathStuff, and then add these static methods to it:
  - sumArray, which accepts an array of integers and returns the sum of the array
  - 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).
- Create another class, TestMyMathStuff, which does not create a MyMathStuff object but still uses each method and prints out the results.
- 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. 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).

```
// modifiers.....          type...  identifier.....          instantiation.
public static              int      identificationNumber = 1;
public static final        int      FAVORITE_NUM         = 49;
```

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, private, 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!

- Print out the value of pi using the static definition in the Math class.
- 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?
- Longer Example: make a small class like Square, Coordinate, or Sphere.
  - Add a static variable named numCreated, and initialize it to zero.
  - 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.
  - 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 (obeying the static thing's public/private visibility). 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

In the previous lab, we learned that:

- an object is the actual value stored in memory
- a reference is a 'handle' or 'address' that tells us where an object is
- a variable is something that may be created to store a reference.

If you have trouble describing the difference between any two of {object, reference, variable}, be sure you grasp the difference sooner than later. Drawing memory diagrams of your program at one specific point in time, and then stating where the variables/references/objects are is a great way to test yourself.

All of this 3-way distinction is for complex values, like objects and arrays – we call all of these sorts of types "reference types". Part of the reasoning is that Java always knows how large an address is (how much space it takes to store one), so it greatly simplifies the process of using these complex values.

Primitive *values* are immutable (can't be changed), so there's no point in distinguishing between a reference to a primitive value and the actual value stored in memory. (Of course a variable storing a primitive value can be changed all you want – as long as you didn't also make it final).

Given this very distinct view between primitive types and reference types, what does Java do when we call methods and use different types of values for our parameters? We'll discuss it in a moment!

Your Turn!
Draw out memory – the variables, references, and objects – at each stage as indicated in the code below. Remember, String is a class, so String values are objects.

```
int x = 5;
String s = "hello";
String t = "goodbye";
// draw out the memory here
t = x+s;
// draw out the memory here
s = t;
// draw out the memory here
```

Perform the same exercise here. This one uses our Square class, as written above.

```
Square s1 = new Square(5);
Square s2 = new Square(8);
Square s3 = s1;
// draw out memory
Square[] ss = new Square[3];
ss[0] = s1;
ss[1] = s2;
ss[2] = s3;
//draw out memory
ss[1].side = ss[2].side;
ss[2].side = 3;
//draw out memory
```

In each of the two above examples, how many were there of:

- variables? (all types)
- references? (all types, all over)
- objects + arrays?

Calling Convention
Each language – be it Java, Python, C, Ada, Algol68, Haskell, PHP, Fortran, or any other – must decide on a calling convention for parameters. Do we copy the entire value that is passed in to a method's parameter? Or do we just pass in a reference to the expression? Do we even evaluate a parameter at all, before calling the method? Let's discuss the issues that help language designers make the decision, and then look at what convention Java follows.

- Primitive values are all small and immutable. It is simple to just send a copy of this tiny value, because we know exactly how much space it takes up, and it is always a very small amount of space.
- Reference types embody information that could vary wildly in size; dealing with wide variety in the space parameters take up (regardless of the number of parameters) could be a real headache

for compiler writers when trying to make method calls efficient, so having small predictable things as parameters is a good thing.

- Should we even evaluate the actual parameters? Migrating the values, expressions, and everything that was in scope for that expression (in order to be able to calculate it later on) is pretty hard, and gives some surprising behaviors. Almost all languages choose to always evaluate the actual parameters (arguments) no matter what, and just deal with these values.
- Copying large values (like a large array) as parameters could be wasteful – imagine writing a method that accepts an array and a key, and the looks for the index at which that key value shows up. Copying an array with thousands or millions of entries in it would be quite wasteful, especially since we don't want to modify it, just inspect it.
- When we do want to modify one small part of a large structure, it is wasteful to copy the entire thing, create a new copy of all that structure with the small change made, and then return this large structure to then be copied into the location originally occupied by the original value.
- Java, like many languages, decided to only allow returning one value. If we want to modify multiple things at once, we would have to somehow group all the values together for the return value of a method (like using a tuple, which Java doesn't have built in).

Java passes actual parameters using **call-by-value**. A parameter's specific value is calculated, and a ***copy*** of this value is given to the method being called. Since a copy was received, the original value is not affected. It is worth rephrasing to say that Java uses call-by-value for both primitive types and reference types. But the implications are different:

- For primitive types, the value is immutable and of a fixed size, so there is no way this copy of the value can affect the original value. (There is no notion of updating part of a primitive value – 5 is always worth |||||, and we can't change that).
- For reference types, the copy of the reference can be modified, and the original reference will not be affected. However, both the original reference and the copy of the reference are able to access and modify the information found *at* that address, so the effects of using the copy to modify the object are still visible after the method returns.
    - Thus when we pass a reference type into a function, the formal parameter and the actual parameter become aliases for the same object.
    - Be careful – if we reassign the parameter (like a = expr), the parameter is no longer an alias of what was passed in; thus this change and any further modifications via the parameter are not affecting what was passed in. It's an easy trap to fall into.

Your Turn! Now that we have discussed references and the calling convention of Java, the following code and questions should help you focus on the differences between the objects in memory; the references to those objects; and the variables that hold references to those objects.

Create the following Foo class:

```
public class Foo {
	public int x;
	public String s;
	public Foo (int x, String s){
		this.x = x;
		this.s = s;
	}
	public String toString() {
		return (" Foo(x="+x+",s=\""+s+"\")");
	}
}
```

Consider (and play around with) the following testing class:

```
public class TestFoo {
	public static void main(String[] args) {
		int y = 0;
		Foo f1 = new Foo(1,"A");
		Foo f2 = new Foo(2,"B");

		//problem #1
		m1(y);
		System.out.println(y);

		//problem #2
		m2(f1);
		System.out.println(f1);

		//problem #3
		m1(f1.x);
		System.out.println(f1);

		//problem #4
		f2 = m3(f1);
		System.out.println("#1: "+f1+". #2: "+f2);

		//problem #5
		f2 = f1;
		System.out.println("#1: "+f1+". #2: "+f2);

	}
	public static void m1 (int a) {
		a = 5;
	}
	public static void m2 (Foo f) {
		f.x = 3;
		f.s = "C";
	}
	public static Foo m3 (Foo f) {
		Foo temp = new Foo(4,"D");
		return temp;
	}
}
```

For each of the problems (listed in comments), answer these questions:

- Is a primitive value or reference value passed in?
- What is printed on the next line?
- Were there any aliases during the method call? (if there was a method call at all)
- Were there any aliases after the method call? (or after running the statements, if no call)

Scope and Visibility

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.
- **method -** defined in a class. accessible through the entire class definition.
- **class -** defined in a package. Accessible through the entire package. (Note: we're ignoring inner/nested classes!)

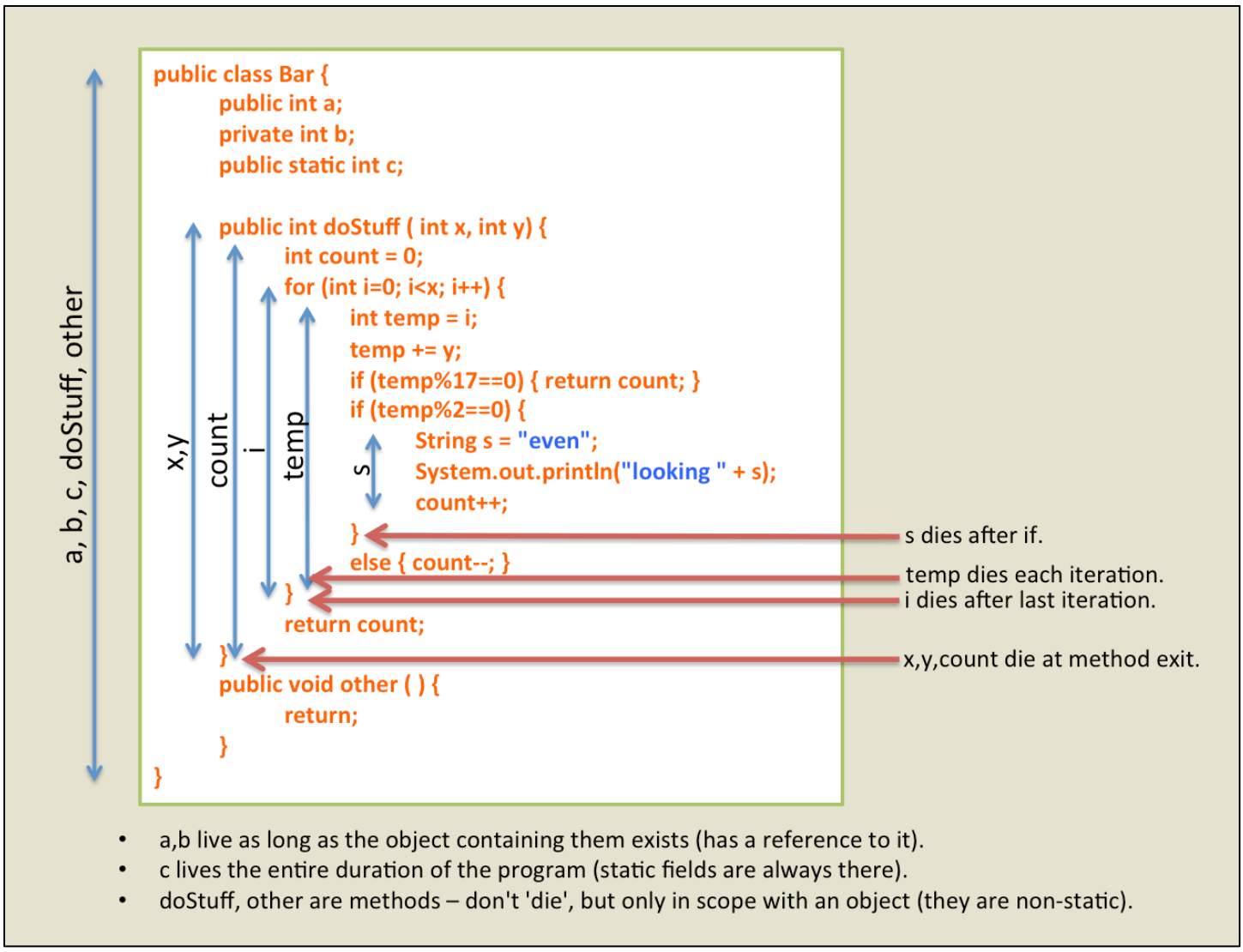
Note: if we have a local variable that is currently referring to an object, and that object has (publicly visible) fields that we can also access, that doesn't mean the interior thing we accessed is in scope here: we used the dot operator to go to a different scope and find something that was in scope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

Lifetimes

- **local variable -** a local definition begins existence at the declaration statement creating it, and dies upon exiting the block of code in which it was created. (The closest-enclosing curly braces { } dictate the lifetime of local definitions). If you declare a variable inside an if-statement, this does in fact mean it is destroyed upon exiting the block.
- **parameter -** a parameter, defined at the start of a method, lives during the entire duration of one call of the method (and dies when we return from the method). Specifically, it is created (and instantiated with the corresponding actual parameter) when the method is entered, and is destroyed when we return from the method.
- **instance variable -** a particular instance variable begins when the object in which it resides begins its existence, and dies when that object is no longer accessible. If you don't give it a starting value in the constructor, Java will complain and not compile your code.
- **class variable -** a class variable begins its existence when the entire program begins running, because the class definition is in scope the entire time the program runs. It doesn't die until the program quits entirely. Note that since no object of the class is required to access it, the definition of its lifetime has nothing to do with the lifetime of any object either.
- **method -** a method exists the entire duration of the class containing it (so during the entire program's duration).
- **class -** a class exists for the entire program's duration.

A collective example is shown in an image below:
→ The **scopes** of things are **written on the left** side
→ destruction points (deaths) are indicated on the right.



Code example (text format):

```
public class Bar { //begin scope of a, b, c, doStuff(), and other()
	public int a;
	private int b;
	public static int c;

	public int doStuff(int x, int y) { //begin scope of x and y
		int count = 0; //begin scope of count
		for (int i = 0; i < x; i++) { //begin scope of i
			int temp = i; //begin scope of temp
			temp += y;
			if(temp % 17 == 0) { return count; }
			if(temp %2 == 0) {
				Strinng s = "even"; //begin scope of s
				System.out.println("looking + s);
				count++;
			} //end scope of s (s dies after if)
		} //end scope i and temp (i dies after last iteration, temp dies each iter.)
	} //end scope of x, y, and count (x, y, count die at method exit)

	public void other() {
		return;
	}
} //end scope of a, b, c, doStuff(), and other()
```

Notes

- a,b live as long as the object containing them exists (has reference to it)
- c lives the entire duration of the program (static fields are always there)
- doStuff() and other() are methods, they don't "die" but are 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:

```
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!

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

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

	public int max (int[] xs) {
		int sum = 0;
		for (int x : xs) {
				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).
- **protected:** used during inheritance. (Spoiler alert: protected members are effectively private except that child classes can access them, unlike actual private members.)
- **default <package-private> :** behaves as public inside the entire package, but private outside the package. (Package-wide visibility). This often leads to bugs once you share your package of code!

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 ());
```

We will talk about **protected** more once we've learned about inheritance. (But to look ahead, you could imagine letting your college kids have access to the fridge, and not have access to your diary; all this, while they have a key to the house that is always locked to outsiders.)

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:

```
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:

```
class BankAccount {
	private int balance;
…}

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. Notice the consistent use of camel-casing.
- There is nothing special about a getter or setter method – Java treats them just like any other public method.

```
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.

Pragmatic Practice: when a definition is extremely simple – in a class that only has state, such as a class representing a pair of ints – if we are going to just offer reading and writing without any extra checks such as disallowing non-negative values, we might as well just make everything public and make the syntax that much more succinct.

Your Turn!

Rectangle Example

- Create a Rectangle class, with fields for length and width. Make them both public initially.
- Make a public constructor. (Note: even constructors can be private! Can you think of when this might be useful?)
- Create or reuse a testing class to create a Rectangle object, then modify its fields, and then print them out.
- Now, realize we want to write good OOP code, and change both fields to be private.
- Add getter and setter methods for both fields. Disallow non-positive values for each field.
- Go back to your testing class and edit the code to use the getters and setters.

- **Going Further:** Let's remove the setter for width (revoke writing privileges for width).
- Add public methods to get and set the perimeter. But don't add a perimeter field!
    - In the setter, re-work the equation 2*(length+width)=perimeter to figure out how to modify **just** the width to achieve the given perimeter.
    - 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.
- 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).

BankAccount Example

- Create a BankAccount class. Give it private fields for accountNum, balance (in pennies), password, and customerName.
- What getters and setters do you want to provide?
- Add a constructor, with parameters for each field.
- Add another constructor, with parameters for each field except balance, and set it to zero. This is an example of an overloaded method.
- 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).
- 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?
- 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.
- Are there any other checks you want to add?
- Test out your code.
- Note on money: Why did we suggest an int instead of a double for money? Although we like to think in terms of dollars, the chance for imprecision in floating point calculations is unnecessary. If we view the amounts as pennies, we can always store the exact value in an int.