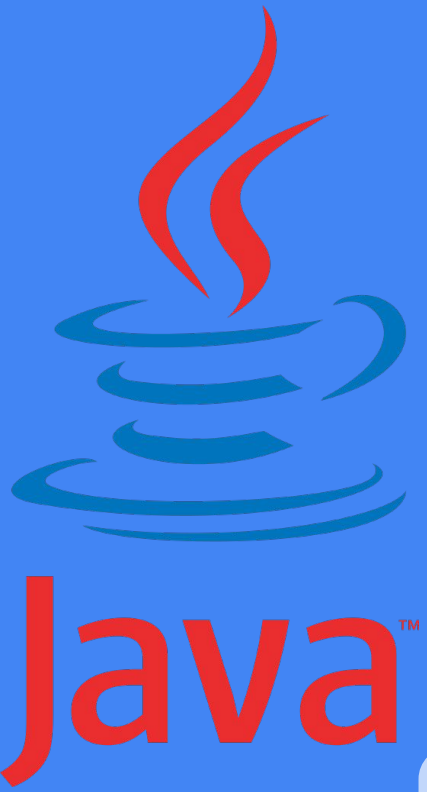
Introduction to Java Programming

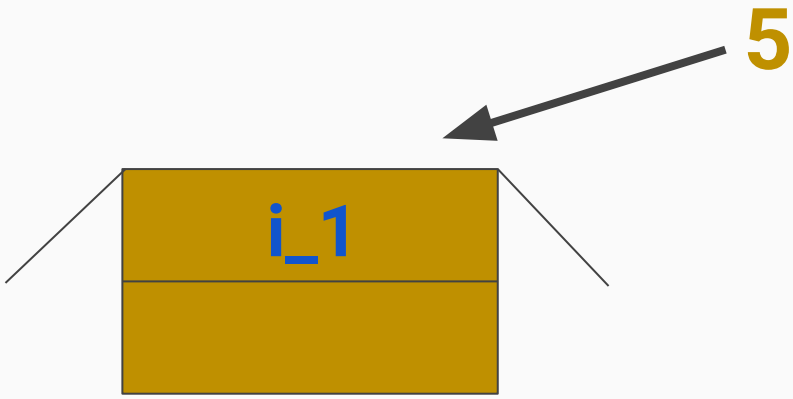
Part 5: OOP Continued - Methods, Inheritance, Scope



REVIEW FROM LAST TIME

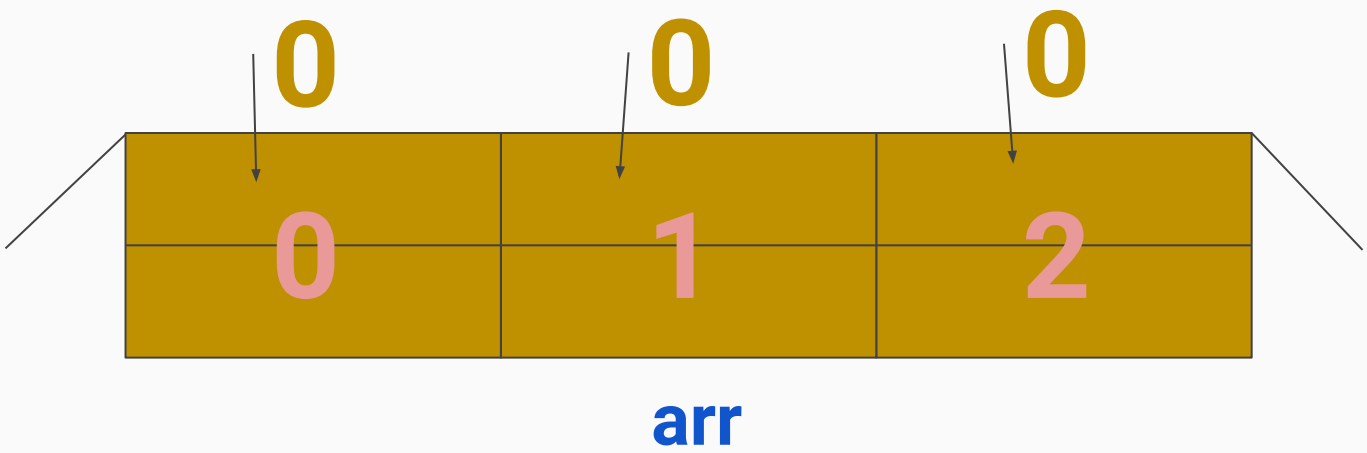
Variable Syntax

```
int i_1 = 5;
```



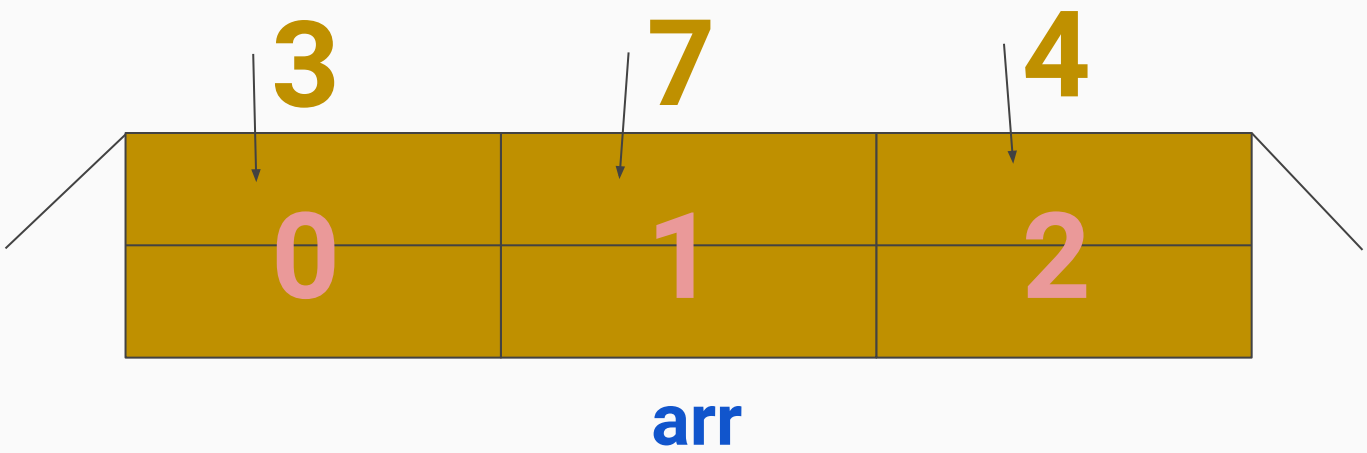
Array Syntax

```
int[] arr = new int[3];
```



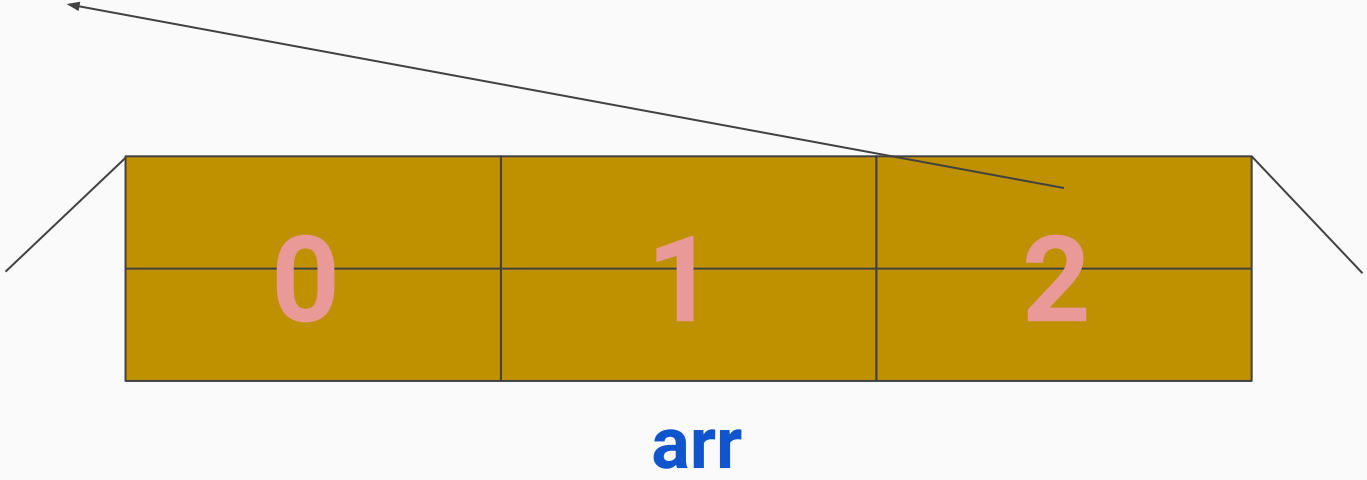
Array Syntax

```
int[] arr = {3,7,4};
```



Accessing Array Elements

```
arr[2];
```



The For, Each Loop:

A “for, each” loop (syntax below) makes iterating over an array or any array-type structure much simpler. This is a common use of a for loop, so it is useful to know this syntax. Note that this doesn’t give you an index variable to work with, however.

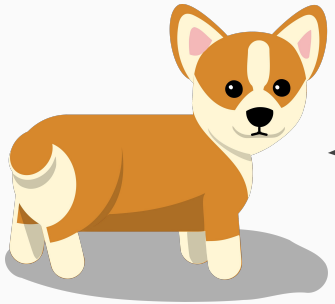
```
for(int o : arr) {
    //code
}
```

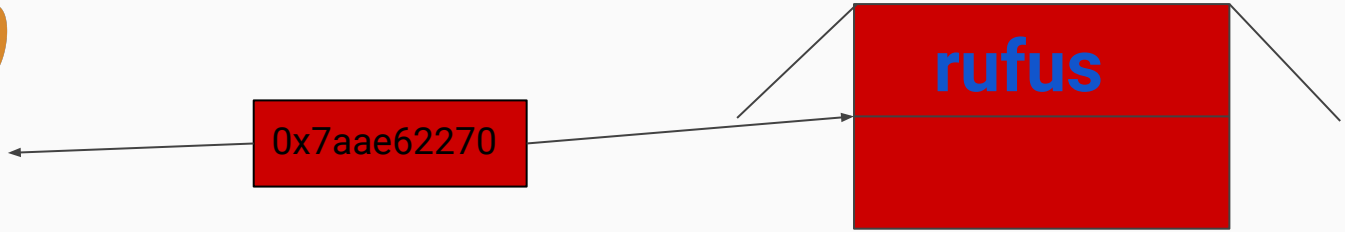
Object Oriented Programming Basics

- A **class** is a file ending in the ".java" extension. All code in Java is contained inside of a **class**. **Classes** provide the code blueprint needed to construct **instances** of themselves, which are called **objects**.
- **Objects** have behaviors and states unique to them. **Classes** are blueprints for objects that define their possible behaviors and states, and allow them to be **constructed**.
- In order to access an object's states (called **fields**) and behaviors (called **methods**), we use a dot (.) after the object's identifier. We have seen this before many times, with Math.max(), or System.out.println()

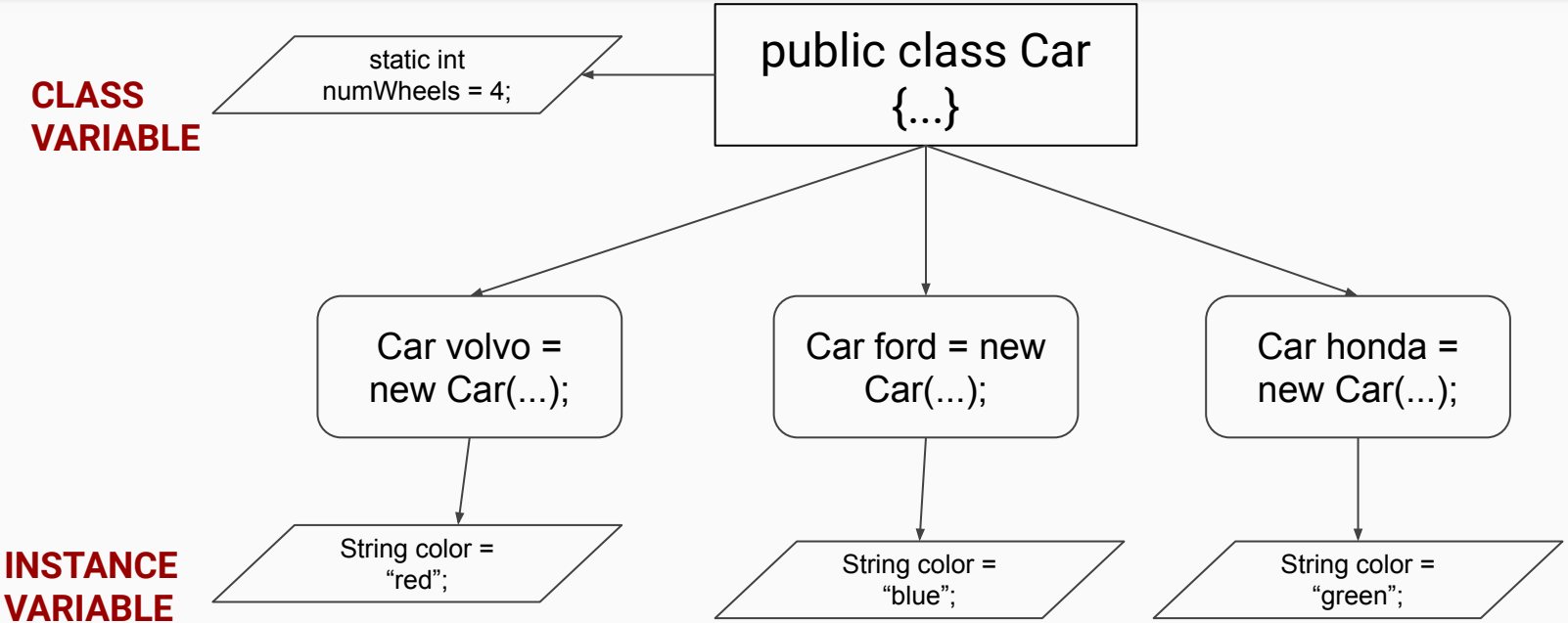
Reference Variable Syntax

```
Dog rufus =
new Dog(7);
```

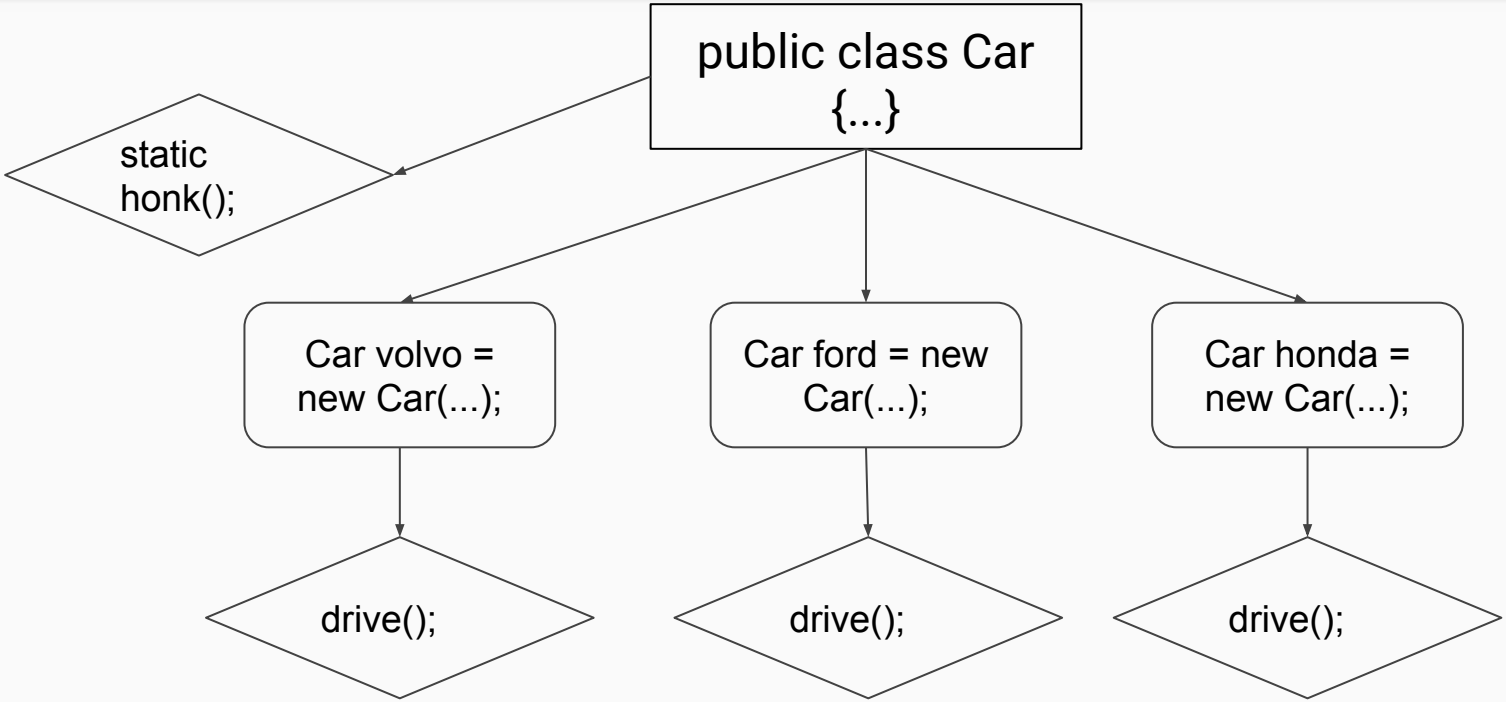




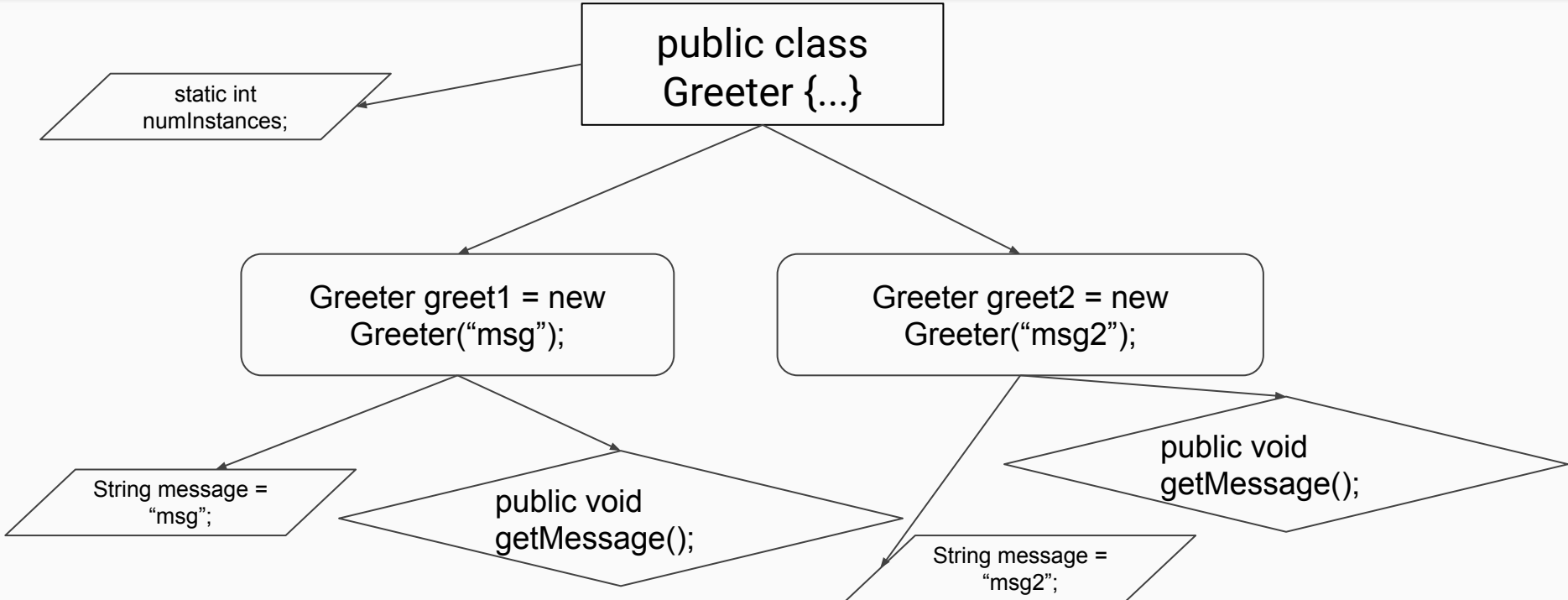
Class Variables vs. Instance Variables



Object Oriented Programming in Java Syntax



Let’s Look at the Greeter Class



END OF REVIEW

Important Access Modifiers

Modifiers are keywords used to define where a field or method can be accessed. They are useful when defining expected and desired behavior for users of a library or set of code. It is good practice to always use the most restrictive modifier possible.

Modifier	Class	Package	Subclass	General
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
Default (no modifier)	Yes	Yes	No	No
private	Yes	No	No	No

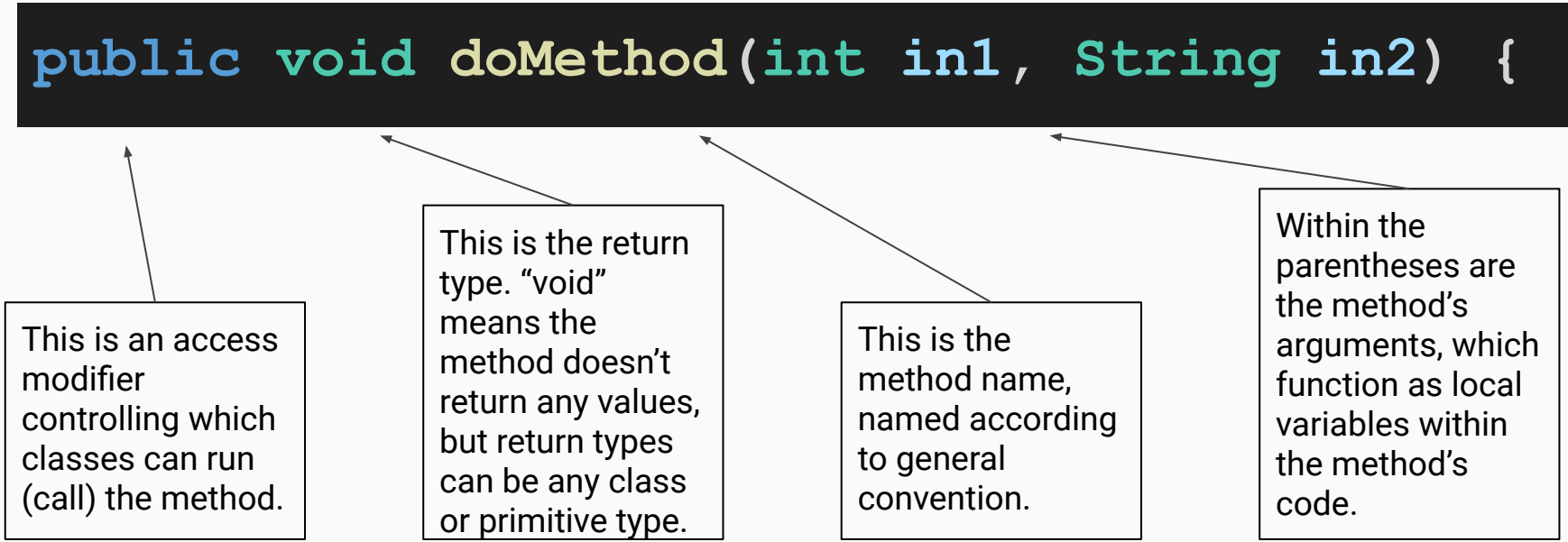
Important Non-Access Modifiers

Modifier	Function
static	Used to declare fields and methods independent of any instance of a class. Used to create **class variables**.
final	Used to indicate a variable can be initialized only once. Effectively, this makes a variable a constant.
abstract	We will use this later - indicates that a class might not provide implementations of its methods and cannot be instantiated.
synchronized, volatile	Used in conjunction with multithreading.

About Methods

- As we have discussed, methods group together code that accomplishes a single, specific function. A method can be run by calling the method name. This can help readability, flexibility, and reduce code repetition. They are recognized by parentheses.
- Methods *belong* to a class. Unless they are static, they cannot be run without an instance of the class.
- Methods take in information (arguments) and output information (return values).
- Variables defined in methods are called **local variables**. All the variables we defined when we were working entirely within *public static void main()* were local variables, as they did not belong to the class.
- Conventionally, method names begin with a lowercase verb. If they contain multiple words, each following word is capitalizedLikeThis.

Method Declaration



What happens to our arguments when we pass them in?

- Java variables are passed in by *value*, not by *reference*. What this essentially means is that when you put a variable as an argument to a method, a copy of whatever is in that variable's "box" is made, and this copy is what is used in the method.
- Remember, reference types contain only a memory address. You can change an object's fields from within a method, but you can't change the memory address that the variable you passed in refers to.

```
public void example(int i, Obj o) {
        i = 5; //Has no effect on the
variable we passed in
        o.changeValue(i); //Can change
an object's fields
        o = new Obj(); //Has no effect
on the variable we passed in
    }
```

A Note on Shadowing Fields

```
public class Test {

    public int field;

    public Test(int field) {
        this.field = field;
    }
}
```

- It is possible for a variable passed in as an argument to a method to have the same name as a field in the class.
- When this happens, the argument variable is said to be “shadowing” the class or instance variable.
- Within this method, using the variable name will always refer to the argument instead of the class’s field.
- To get access to the class’s field, use the **this** keyword.

Method Overloading

- Multiple methods can have the same name if their arguments are different. This is known as method overloading.
- Sometimes, this can make code ambiguous to use and understand, so it should be avoided if possible. However, it is a useful tool in some cases.

```
public void doThing(int i) {

}

public void doThing(String s) {

}
```

Programming Tricks: Recursion

- Recursion is a programming trick in which a method calls itself during its own execution.
- This may seem weird at first, but if you modify some sort of variable in a similar way as you would a loop, you can get a single method to behave in a similar way as a for loop or a while loop.
- Let's take a look at how this works!

```
public void doThing(int i) {
    doThing(i + 1);
}
```

Project: A Recursive Counter

Encapsulation

```
    //Define a setter method
    public void setMessage(String str)
{
        this.message = str;
    }

    //Define a getter method
    public String getMessage() {
        return this.message;
    }
```

- Encapsulation is an object oriented programming concept that essentially dictates that all instance variables should be declared private, and getter and setter methods should be used to control access to them.
- This makes it easier for others to use your code - there is no ambiguity about whether they are misusing your variables.

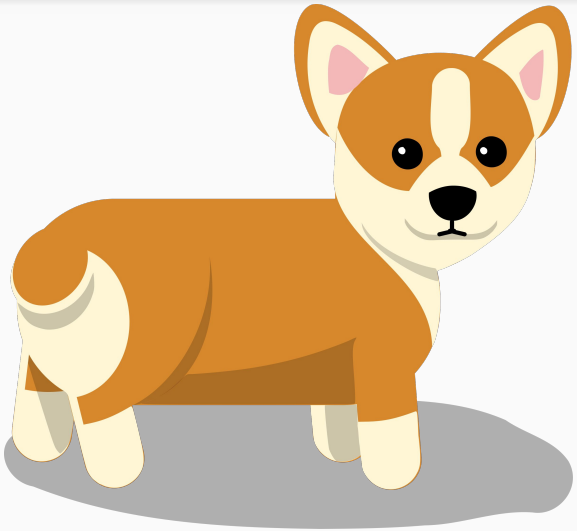
Variable Scope

- We have talked about access modifiers, which define which classes can access another class's fields or methods.
- What about local variables and method arguments? Where can we access variables?
- The **scope** of a variable refers to where the variable can be accessed, and is determined by where in your code it was declared. The rules are:

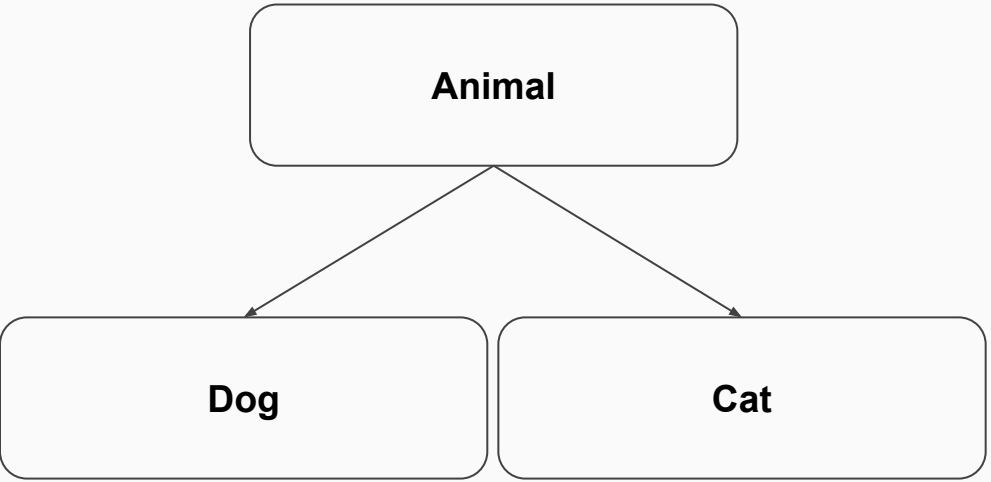
- A variable declared within a class but outside of any method is known as a member variable (including both instance and class variables), and can be accessed from any method within the class. It can be accessed from outside the class according to its access modifier.
- A variable declared within a method is local to that method only, and ceases to exist when the method's execution is over.
- A variable declared within a block (such as an if statement or while loop) exists only within that block.

An Introduction to Inheritance

- Suppose we have to program a representation of cats and dogs that can walk, eat, and make noise. What is the most efficient way we could do this?
- So far, it would seem like the best way to do this would be to create two separate classes. Cats and dogs have different states and behaviors, so it would make sense for them to be represented by different classes, right?



An Introduction to Inheritance



- You might notice that cats and dogs share a lot of behaviors and states in common - they both eat, walk, have four legs and a tail - and these similarities translate to code repetition.
- In general, code repetition is inefficient and undesirable. It makes code harder to read.
- **What if we introduced a third class that handled all the behaviors cats and dogs had in common?**

An Introduction to Inheritance

- Java has a feature that lets us do this! It is called **inheritance**. One class can inherit from one other class. The class that inherits the other is called the subclass, while the class the subclass inherits from is called the superclass.
- A subclass has the same fields and methods as its superclass, but can only access them if their access level is protected or public.
- This allows us to share behaviors between similar classes. Interesting note - every single class in Java inherits from the Object class.
- Every subclass has its own instance of its superclass, which can be accessed with the **super** keyword. This functions similarly to the **this** keyword.
- Let's try this out!

Project: Animals, Cats, and Dogs

If There’s Time:

Why would an interface be a better choice?