Chapter-4:
How Objects Behave

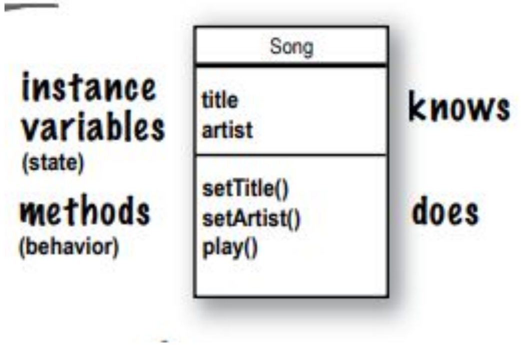
Upcode Software
Engineer Team

-2023-

CONTENT

What is **Object Behave** ? (1/n)

- A class is the blueprint for an object.
- When you write a class, you're describing how the JVM should make an object of that type.
- You already know that every object of that type can have different instance variable values.



Remember: a class describes what an object knows and what an object does

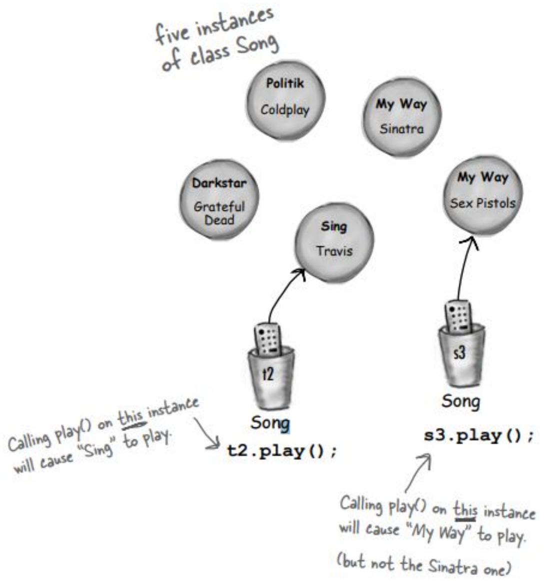
What is **Object Behave** ? (2/n)

Can every object of that type have different method behavior?

- Every instance of a particular class has the same methods, but the methods can behave differently based on the value of the instance variables.

```
void play() {
soundPlayer.playSound(title);
}
```

```
Song t2 = new Song();
t2.setArtist("Travis");
t2.setTitle("Sing"); Song s3 = new
Song(); s3.setArtist("Sex Pistols");
s3.setTitle("My Way");
```


five instances of class Song
Politik
Coldplay
My Way
Sinatra
Darkstar
Grateful Dead
Sing
Travis
My Way
Sex Pistols
t2
s3
Song
Song
Calling play() on this instance will cause "Sing" to play.
t2.play();
s3.play();
Calling play() on this instance will cause "My Way" to play. (but not the Sinatra one)

You can send and return objects to a method. (1/n)

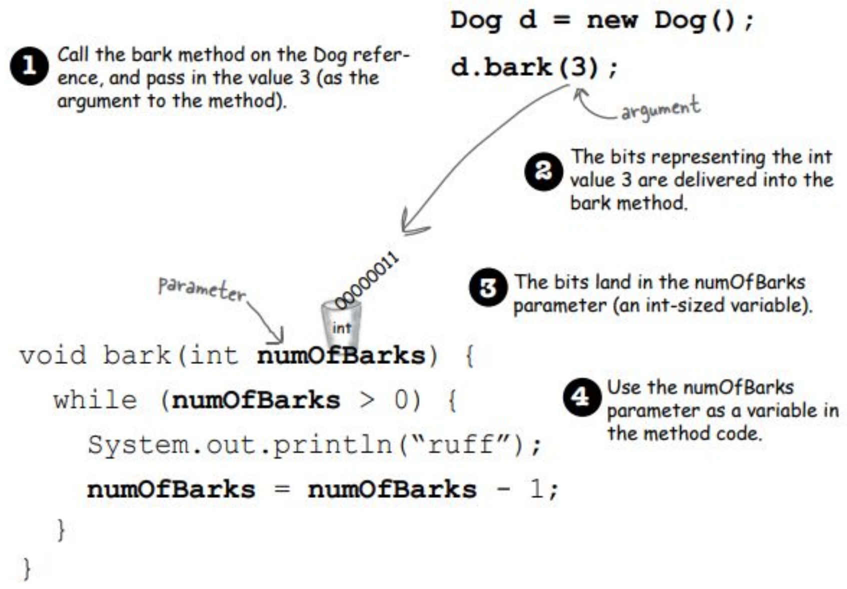
- Just as you expect from any programming language, you can pass values into your methods.
- Depending on your programming background and personal preferences, you might use the term arguments or perhaps parameters for the values passed into a method.
- So you can call them whatever you like (arguments, donuts, hairballs, etc.) but we’re doing it like this:

  A method uses parameters. A caller passes arguments.

- Arguments are the things you pass into the methods.
- And a parameter is nothing more than a local variable. A variable with a type and a name, that can be used inside the body of the method.

You can send and return objects to a method. (2/n)

- But here's the important part: If a method takes a parameter, you must pass it something. And that something must be a value of the appropriate type.



```java
Dog d = new Dog();
d.bark(3);
```

1 Call the bark method on the Dog reference, and pass in the value 3 (as the argument to the method).

2 The bits representing the int value 3 are delivered into the bark method.

3 The bits land in the numOfBarks parameter (an int-sized variable).

4 Use the numOfBarks parameter as a variable in the method code.

```java
void bark(int numOfBarks) {
  while (numOfBarks > 0) {
    System.out.println("ruff");
    numOfBarks = numOfBarks - 1;
  }
}
```

You can send and return objects to a method. (3/n)

You can get things back from a method.

- Methods can return values. Every method is declared with a return type, but until now we've made all of our methods with a void return type, which means they don't give anything back.

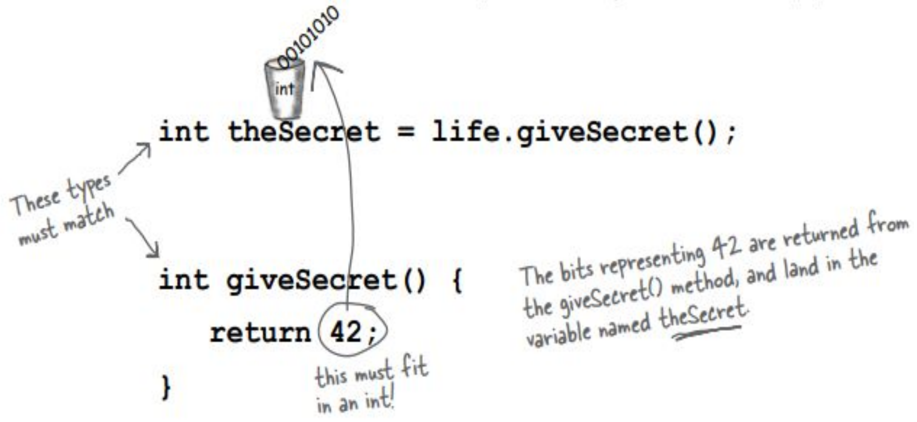
```
void go() { }
```

- But we can declare a method to give a specific type of value back to the caller, such as:

```
int giveSecret() {
return 42;
}
```

You can send and return objects to a method. (3/n)

- If you declare a method to return a value, you must return a value of the declared type! (Or a value that is compatible with the declared type.
- **Whatever you say you'll give back, you better give back!**



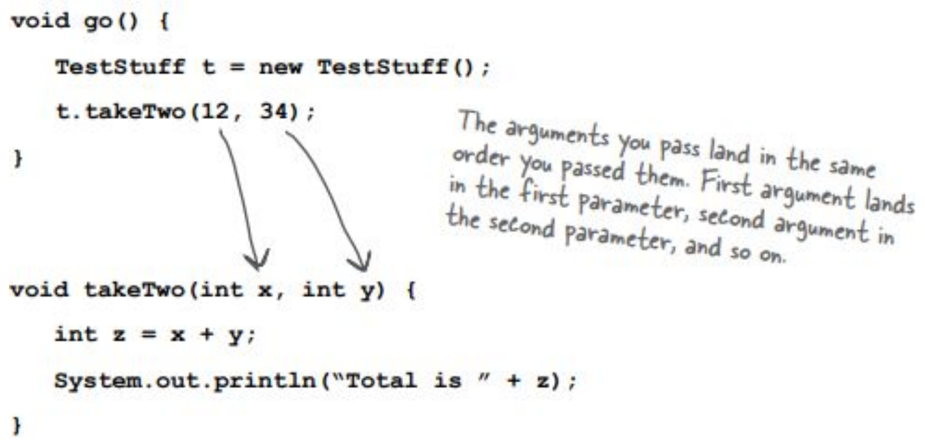
```
int theSecret = life.giveSecret();

int giveSecret() {
    return 42;
}
```

00101010
int
These types must match
this must fit in an int!
The bits representing 42 are returned from the giveSecret() method, and land in the variable named theSecret.

The compiler won't let you return the wrong type of thing.

You can send more than one thing to a method. (1/n)

- Methods can have multiple parameters. Separate them with commas when you declare them, and separate the arguments with commas when you pass them. Most importantly, if a method has parameters, you must pass arguments of the right type and order.
- **Calling a two-parameter method, and sending it two arguments.**

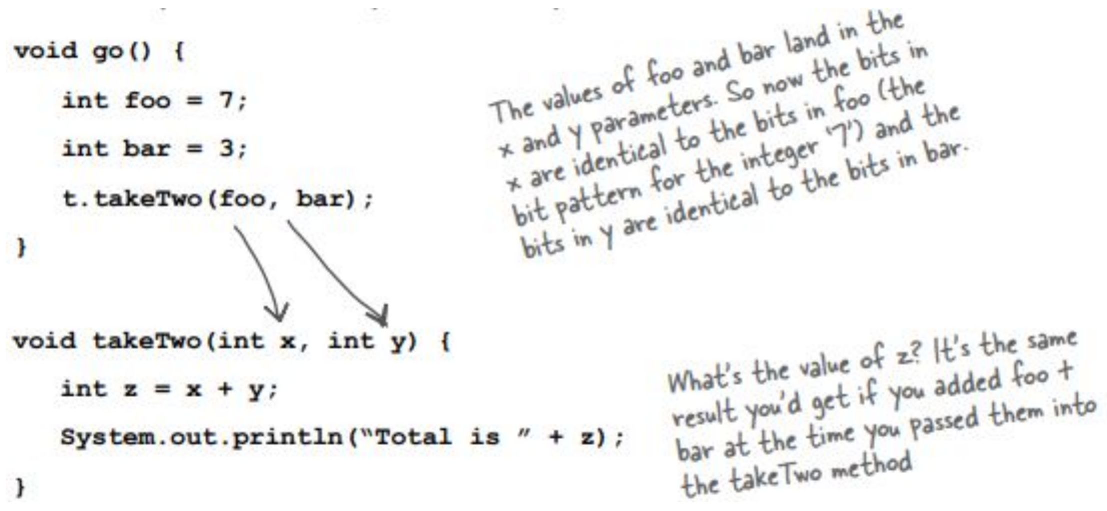


```
void go() {
   TestStuff t = new TestStuff();
   t.takeTwo(12, 34);
}


void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```

The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

You can send more than one thing to a method. (2/n)

- You can pass variables into a method, as long as the variable type matches the parameter type.



```
void go() {
    int foo = 7;
    int bar = 3;
    t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
    int z = x + y;
    System.out.println("Total is " + z);
}
```

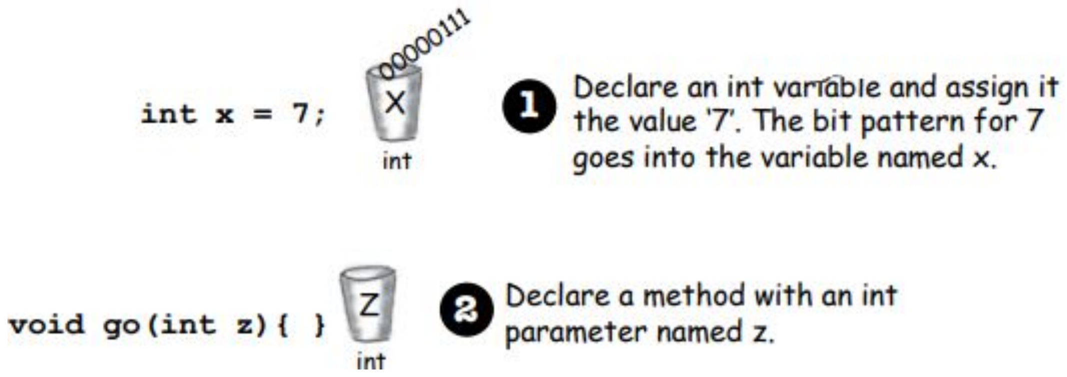
The values of foo and bar land in the x and y parameters. So now the bits in x are identical to the bits in foo (the bit pattern for the integer '7') and the bits in y are identical to the bits in bar.

What's the value of z? It's the same result you'd get if you added foo + bar at the time you passed them into the takeTwo method

You can send more than one thing to a method. (3/n)

Java is pass-by-value.

That means pass-by-copy



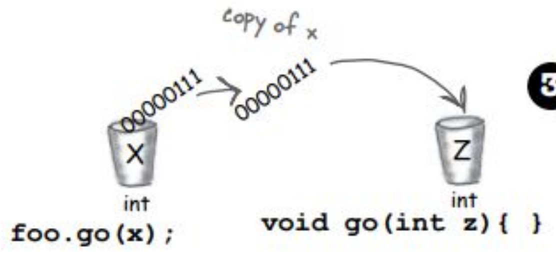
```
int x = 7;
```

1 Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.

```
void go(int z){ }
```

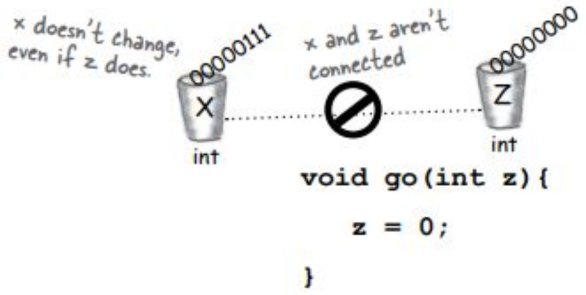
2 Declare a method with an int parameter named z.

You can send more than one thing to a method. (4/n)



```
foo.go(x);      void go(int z){ }
```

❸ Call the go() method, passing the variable x as the argument. The bits in x are copied, and the copy lands in z.



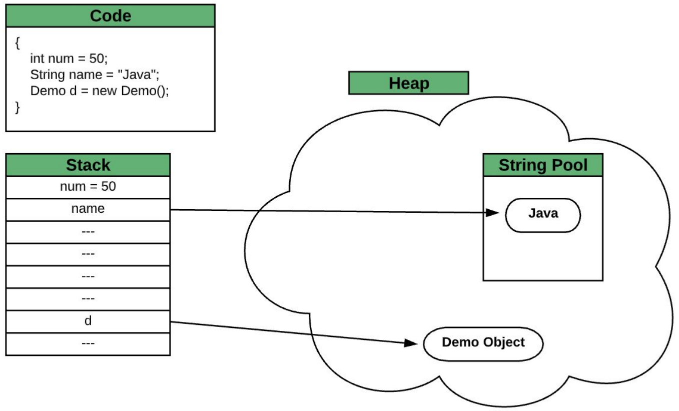
```
void go(int z){
    z = 0;
}
```

❹ Change the value of z inside the method. The value of x doesn't change! The argument passed to the z parameter was only a copy of x.

The method can't change the bits that were in the calling variable x.

What is String pool ? (1/n)

- **String pool** is nothing but a storage area in Java heap where string literals stores. It is also known as **String Intern Pool** or **String Constant Pool**. It is just like object allocation. By default, it is empty and privately maintained by the **Java String** class.
- Whenever we create a string the string object occupies some space in the heap memory. Creating a number of strings may increase the cost and memory too which may reduce the performance also.



```
{
    int num = 50;
    String name = "Java";
    Demo d = new Demo();
}
```

What is String pool ? (2/n)

- The JVM performs some steps during the initialization of string literals that increase the performance and decrease the memory load. To decrease the number of String objects created in the JVM the String class keeps a pool of strings.
- When we create a string literal, the JVM first check that literal in the String pool. If the literal is already present in the pool, it returns a reference to the pooled instance. If the literal is not present in the pool, a new String object takes place in the String pool.

What is String pool ? (3/n)

- There are two ways to create a string in Java:
- Using String Literal.

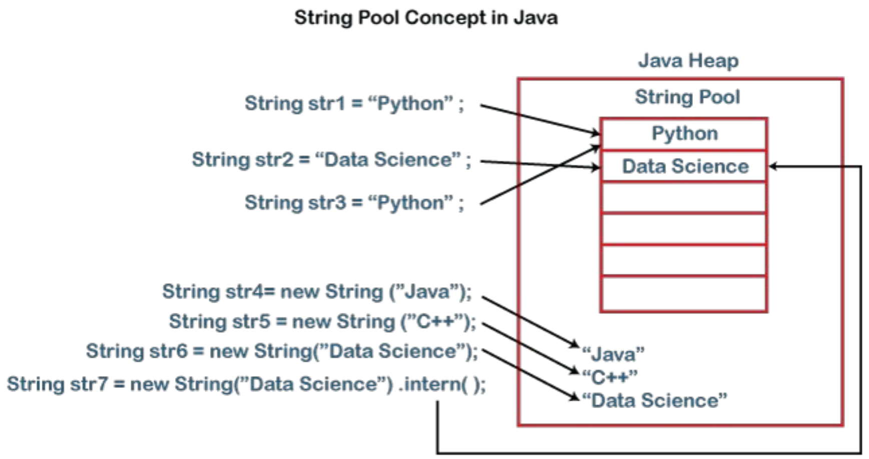
```
String str1 = "Python";

String str2 = "Data Science";

String str3 = "Python";
```

- Using new Keyword.

```
String str1 = new String ("Java");

String str2 = new String ("C++");

String str3 = new String ("Data Science");
```

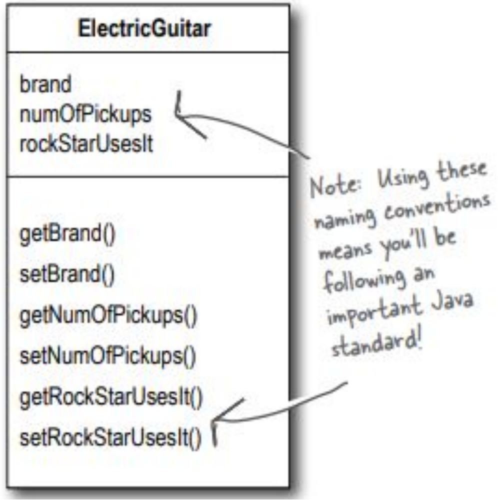
What is String pool ? (4/n)

- Let's understand what is the difference between them. Let's compare the string literals' references.
- s1==s3 //true
- s2==s3 //false


String Pool Concept in Java
Java Heap
String Pool
String str1 = "Python" ;
String str2 = "Data Science" ;
String str3 = "Python" ;
Python
Data Science
String str4= new String ("Java");
String str5 = new String ("C++");
String str6 = new String("Data Science");
String str7 = new String("Data Science") .intern( );
"Java"
"C++"
"Data Science"

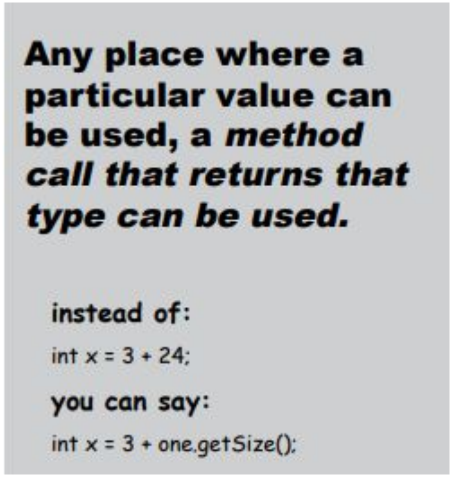
Encapsulation in an object. (1/n)

- **Cool things you can do with parameters and return types.**
- Getters and Setters. If you're into being all formal about it, you might prefer to call them Accessors and Mutators. But that's a waste of perfectly good syllables.
- Besides, Getters and Setters fits the Java naming convention, so that's what we'll call them.
- Getters and Setters let you, well, get and set things.



Encapsulation in an object. (2/n)

- Even though the methods don't really add new functionality, the cool thing is that you can change your mind later. you can come back and make a method safer, faster, better.



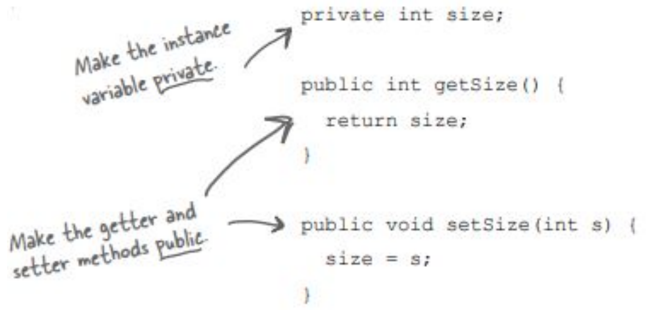
Any place where a particular value can be used, a *method call that returns that type can be used.*

instead of:

int x = 3 + 24;

you can say:

int x = 3 + one.getSize();

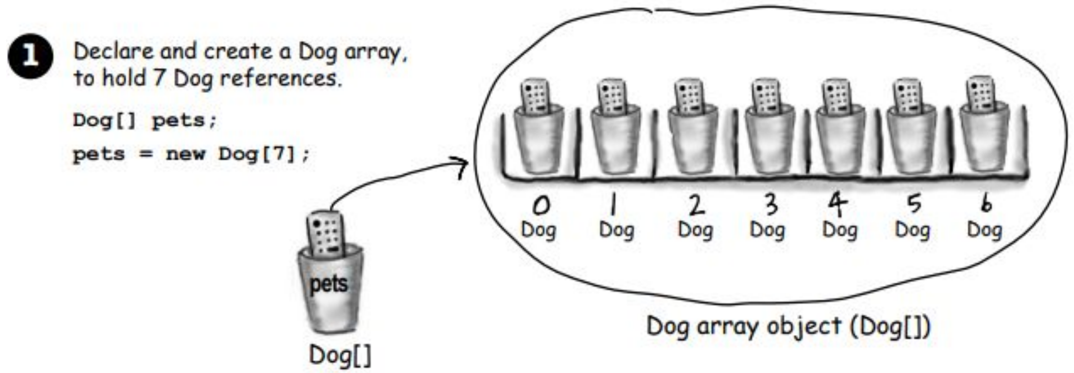


Make the instance variable private.
Make the getter and setter methods public.

```
private int size;

public int getSize() {
  return size;
}

public void setSize(int s) {
  size = s;
}
```

How do objects in an array behave ? (1/n)

- Just like any other object. The only difference is how you get to them. In other words, how you get the remote control. Let's try calling methods on Dog objects in an array.



```
Dog[] pets;
pets = new Dog[7];
```

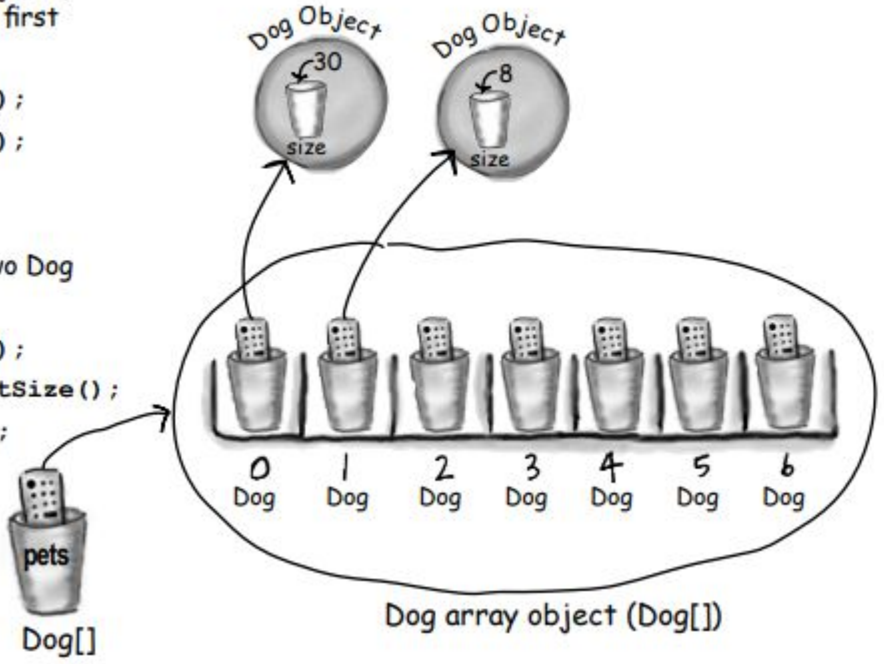
How do objects in an array behave ? (2/n)

2 Create two new Dog objects, and assign them to the first two array elements.

```
pets[0] = new Dog();
pets[1] = new Dog();
```

3 Call methods on the two Dog objects.

```
pets[0].setSize(30);
int x = pets[0].getSize();
pets[1].setSize(8);
```



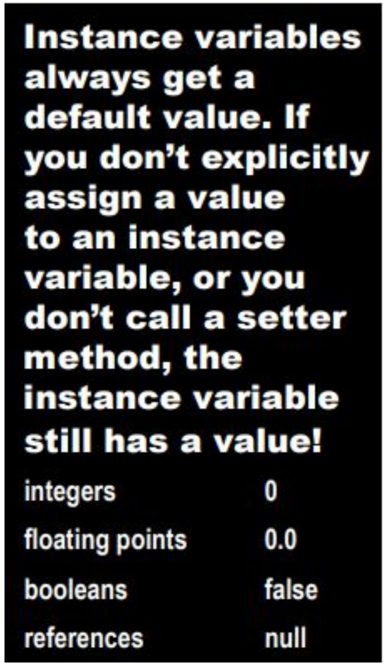
Declaring and initializing instance variables.

- You already know that a variable declaration needs at least a name and a type:

```
int size;
String name;
```

- And you know that you can initialize (assign a value) to the variable at the same time:

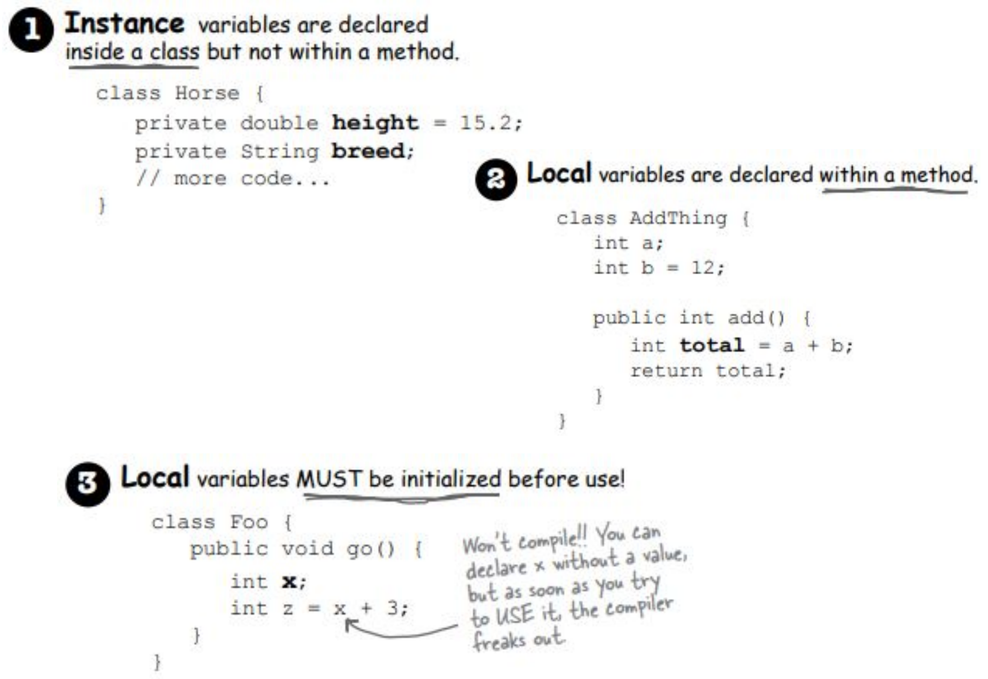
```
int size = 420;
String name = “Donny”;
```



Instance variables always get a default value. If you don’t explicitly assign a value to an instance variable, or you don’t call a setter method, the instance variable still has a value!

integers	0
floating points	0.0
booleans	false
references	null

The difference between instance and local variables.



1 **Instance** variables are declared inside a class but not within a method.

```
class Horse {
   private double height = 15.2;
   private String breed;
   // more code...
}
```

2 **Local** variables are declared within a method.

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

3 **Local** variables MUST be initialized before use!

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

Local variables do **NOT** get a default value! The compiler complains if you try to use a local variable before the variable is initialized.

Reference

1. Head First book (page 71 - 88)
2. Website

Thank you!

Presented by **Jahongir Sherjonov**

(jakhongirsherjonov@gmail.com)