Java Review – Know Your Variables

- Variables come in two categories: **primitive and reference.**
- Variables can be used as object **state** (instance variables), and as **local variables** (variables declared within a *method*).
- Variables can be used as **arguments** (values sent to a method by the calling code), and as **return types** (values sent back to the caller of the method).

In this lecture we will review Java types and look at what you can *declare* as a variable, what you can *put* in a variable, and what you can *do* with a variable.

Declaring a variable

- **Java cares about type**.
- You must declare the type of your variable.
- Variables come in two categories: ***primitive*** and ***reference***.
  - Primitives hold fundamental values, including integers, booleans, and floating point numbers.
  - Refrences hold refrences to objects.

Declaring a variable

Variables must have a type.

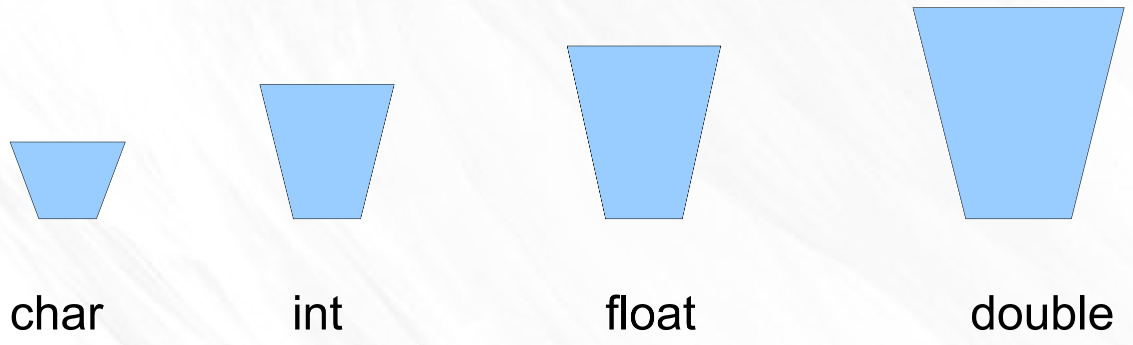
Variables must have a name.

type	name
int	count;
long	interestRate;
float	salary;
double	gasPrice;
boolean	isAVowel;
char	letter;
Dog	dog;

Initializing a variable

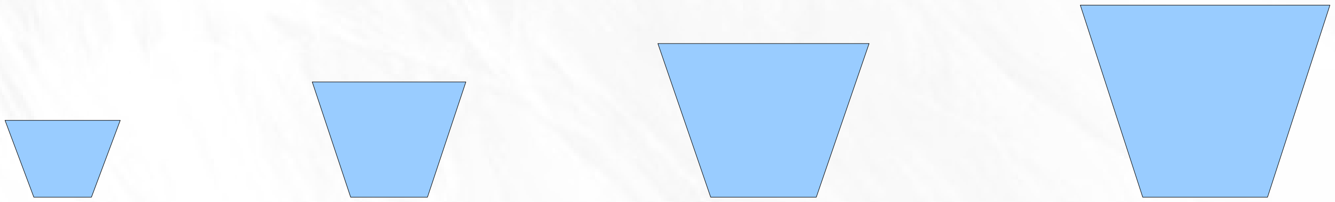
A variable is just a cup. A container. It *holds* something.

Primitives are like cups at a coffee shop. They come in different sizes and each has a different name.



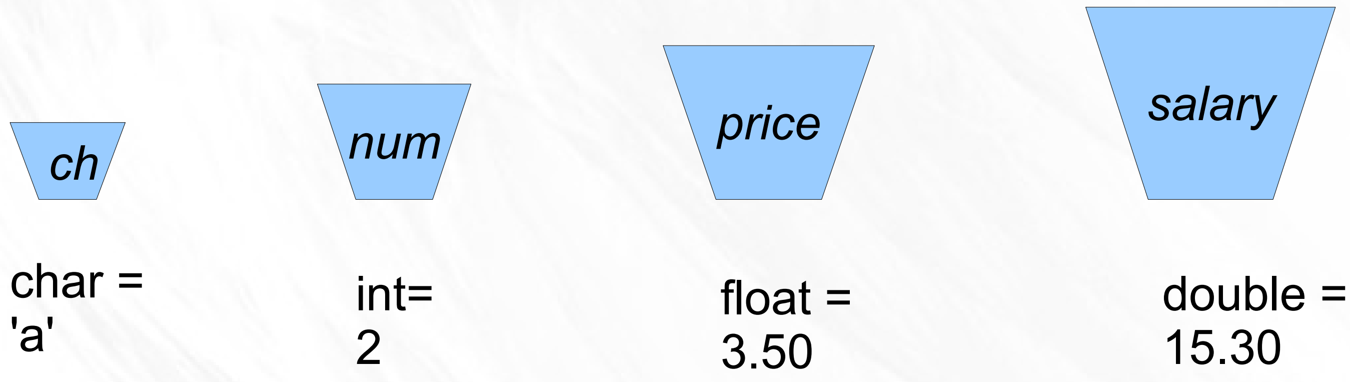
Initializing a variable

Each cup holds a value of it's *type*. Each cup needs to have its own *name*.



Initializing a variable

Each cup holds a value of it's *type*. Each cup needs to have its own *name*.



Initializng a variable

Be sure the value can fit into the variable.

int x = 24;

byte b = x;

// won't work (int data type is *larger* than byte data type)

However,

byte b = 24;

int x = b;

//will work (you can fit a *smaller* data type into a larger data type)

Initializing a variable

You can assign a value to a variable in one of several ways including:

- Type a *literal* value after the equals sign:

  (x = ***12***, isGood = ***true***, etc.)

- Assign the value of one variable to another

  (x = y)

- Use an expression combining the two

  (x = y + ***43***)

Initializing a variable

But what about reference variables?

- There is actually no such thing as an **object** variable, there is only an object **reference** variable.
- An object reference variable holds bits that represent a way to *access* an object.
- It does not hold the object itself, but it holds something like a pointer. Or an address. Except, in Java we don't really know *what* is inside a reference variable. We *do* know that whatever it is, it represents one and only one object. And the JVM knows how to use the reference to get the object.

Object references

- Although a primitive variable is full of bits representing the actual ***value*** of the variable, an object reference variable is full of bits representing ***a way to get to the object***.
- You use the dot operator (.) on a reference variable to say "use the thing *before* the dot to get me the thing *after* the dot."

For example:

myDog.bark( ) ;

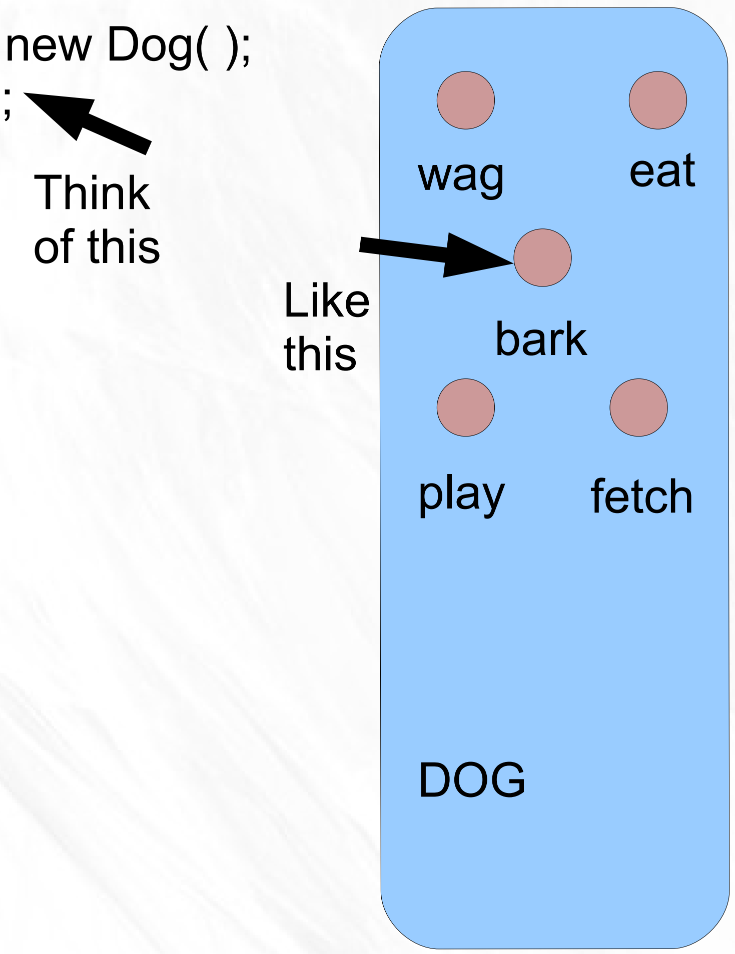
Means, "use the object referenced by the variable 'myDog' to invoke the bark( ) method."

When you use the dot operator on an object reference variable, think of it like pressing a button on the remote control for that object.

Object references

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

Think of this

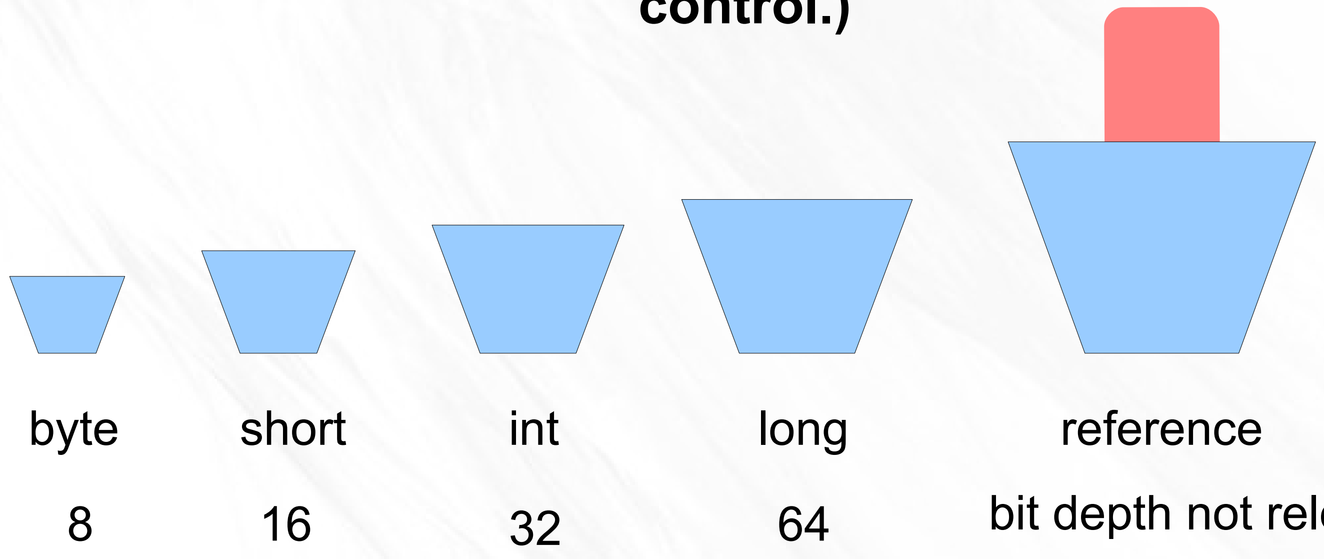


Think of a Dog **reference** variable as a Dog **remote control**. You use it to get object to do something (invoke methods).

Object references

An object reference is just another variable value.

(Something that goes in a cup. Only this time the value is a remote control.)

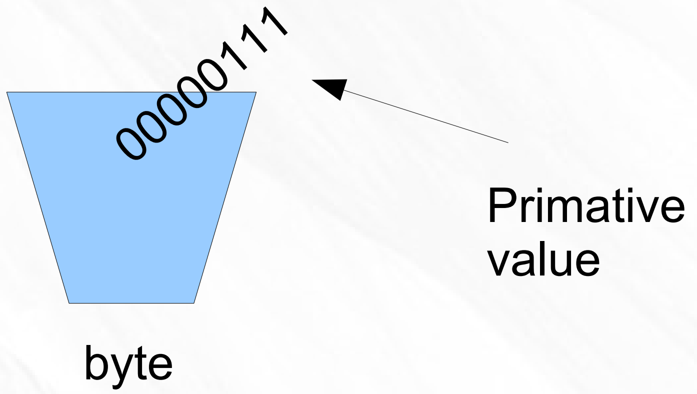


Object refrences

Primitive Variable

byte x = 7;

The bits representing 7 go into the variable. (00000111).



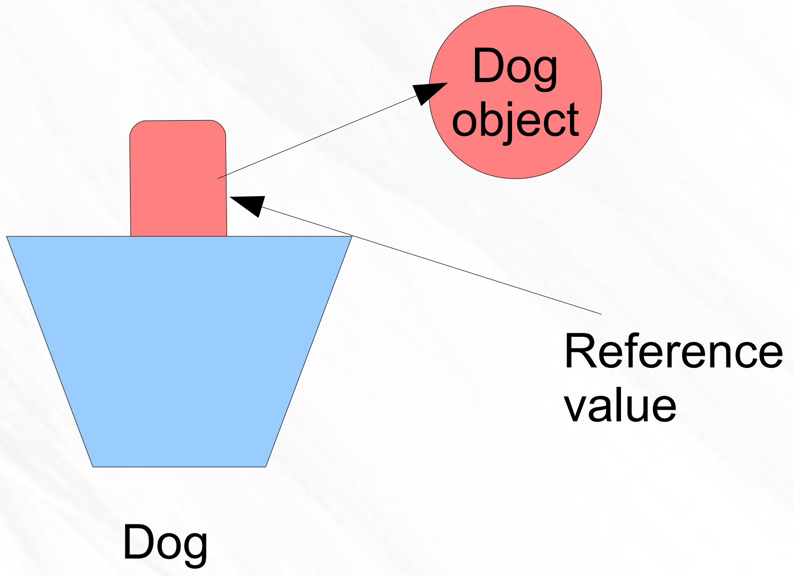
Object references

Reference Variable

Dog myDog = new Dog( ) ;

The bits representing a way to get to the Dog object go into the variable.

The Dog object itself does not go into the variable!



Object references

- With primitive variables the value of the variable is... *the value* [5, -26.7, 'a'].
- With reference variables, the value of the variable is... *bits representing a way to get to a specific object*.
- You don't know (or care) how any particular JVM implements object references.

Object references

The 3 steps of object declaration, creation and assignment

$$\overbrace{\texttt{Dog myDog}}^{1} \;\overset{3}{=}\; \overbrace{\texttt{new Dog( );}}^{2}$$

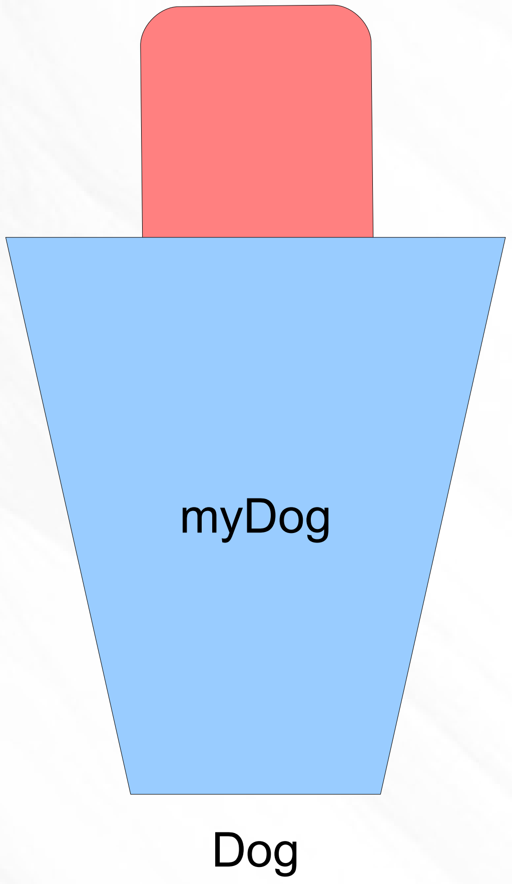
1. Declare a reference variable
2. Create an object
3. Link the object and the reference

Object references

1. Declare a reference variable

Dog myDog = new Dog( );

Tells the JVM to allocate space for a reference variable, and names that variable *myDog*. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or Bird or a Snake.

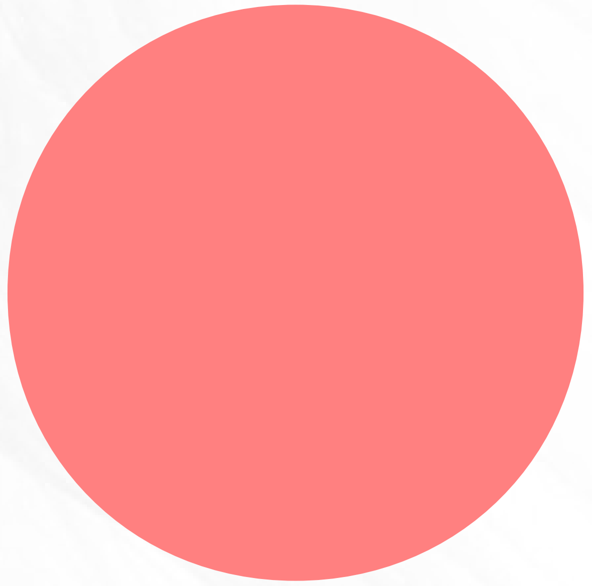


Object references

2. Create an object

`Dog myDog =` `new Dog( );`

Tells the JVM to allocate space for a new Dog object on the garbage heap.



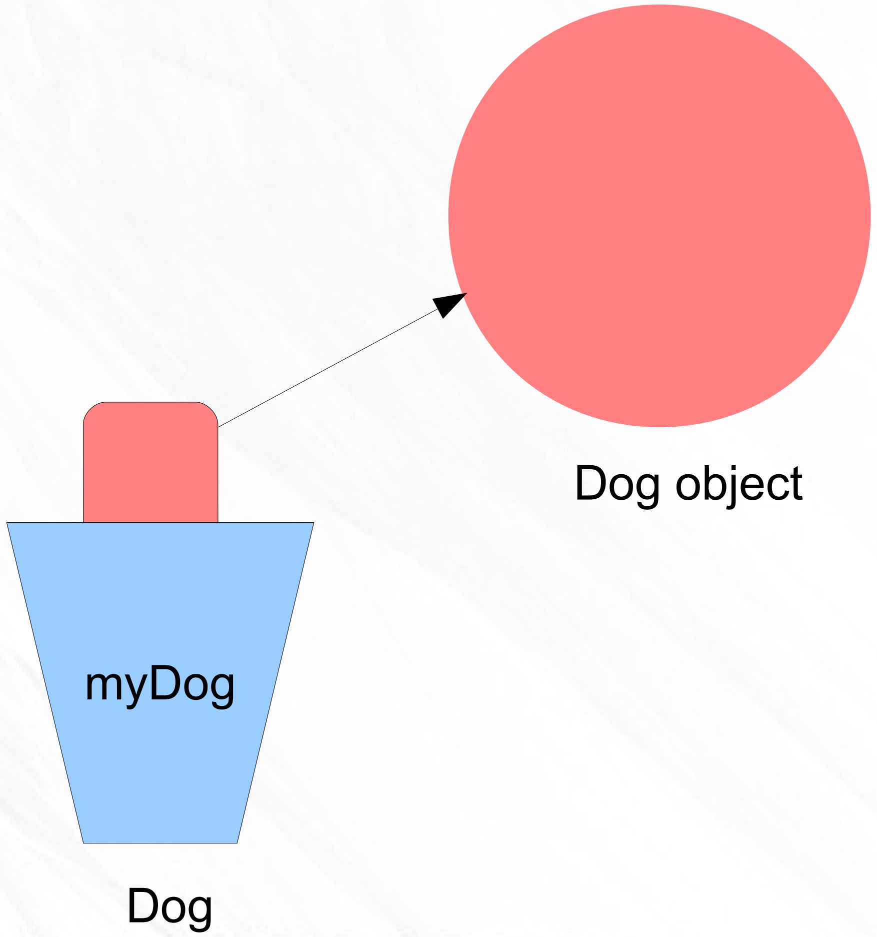
Dog object

Object references

3. Link the object and the reference

```
Dog myDog = new Dog ( );
```

Assigns the new Dog to the reference variable myDog. In other words, ***programs the remote control.***

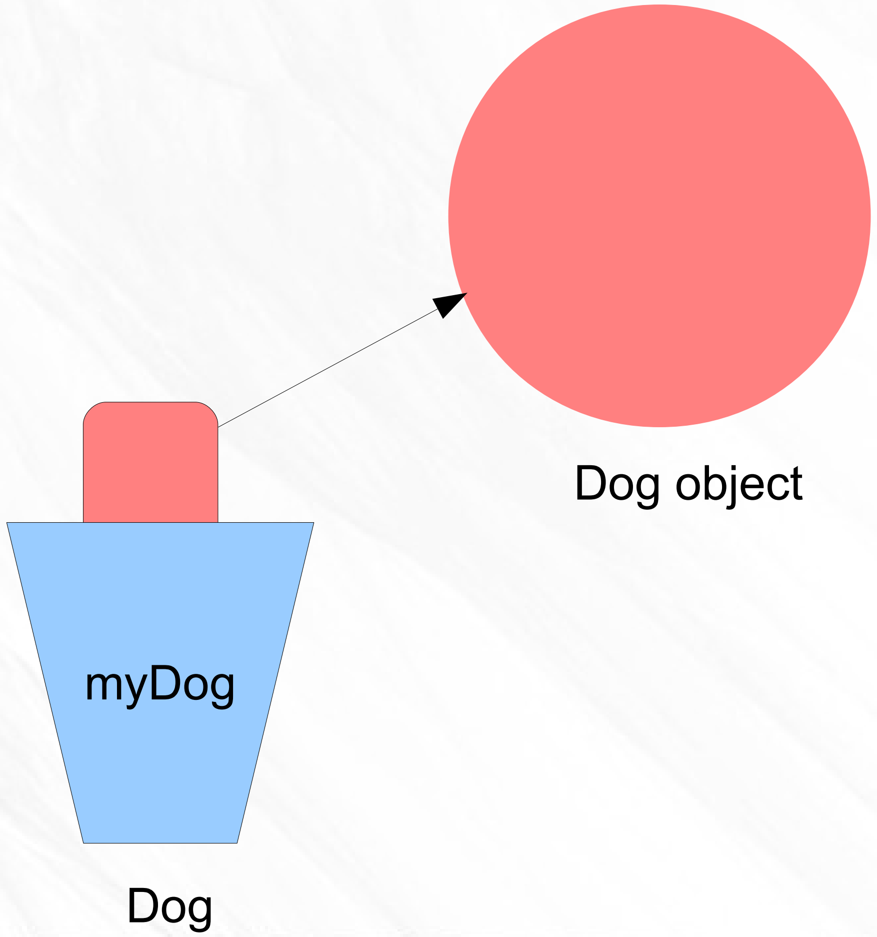


Object references

3. Link the object and the reference

```
Dog myDog = new Dog ( );
```

Assigns the new Dog to the reference variable myDog. In other words, ***programs the remote control.***



Object refrences

What is the garbage collection heap?

The garbage collector is a program which runs on the Java Virtual Machine which gets rid of objects which are not being used by a Java application anymore. It is a form of automatic memory management.

Object references

Life on the garbage-collectible heap

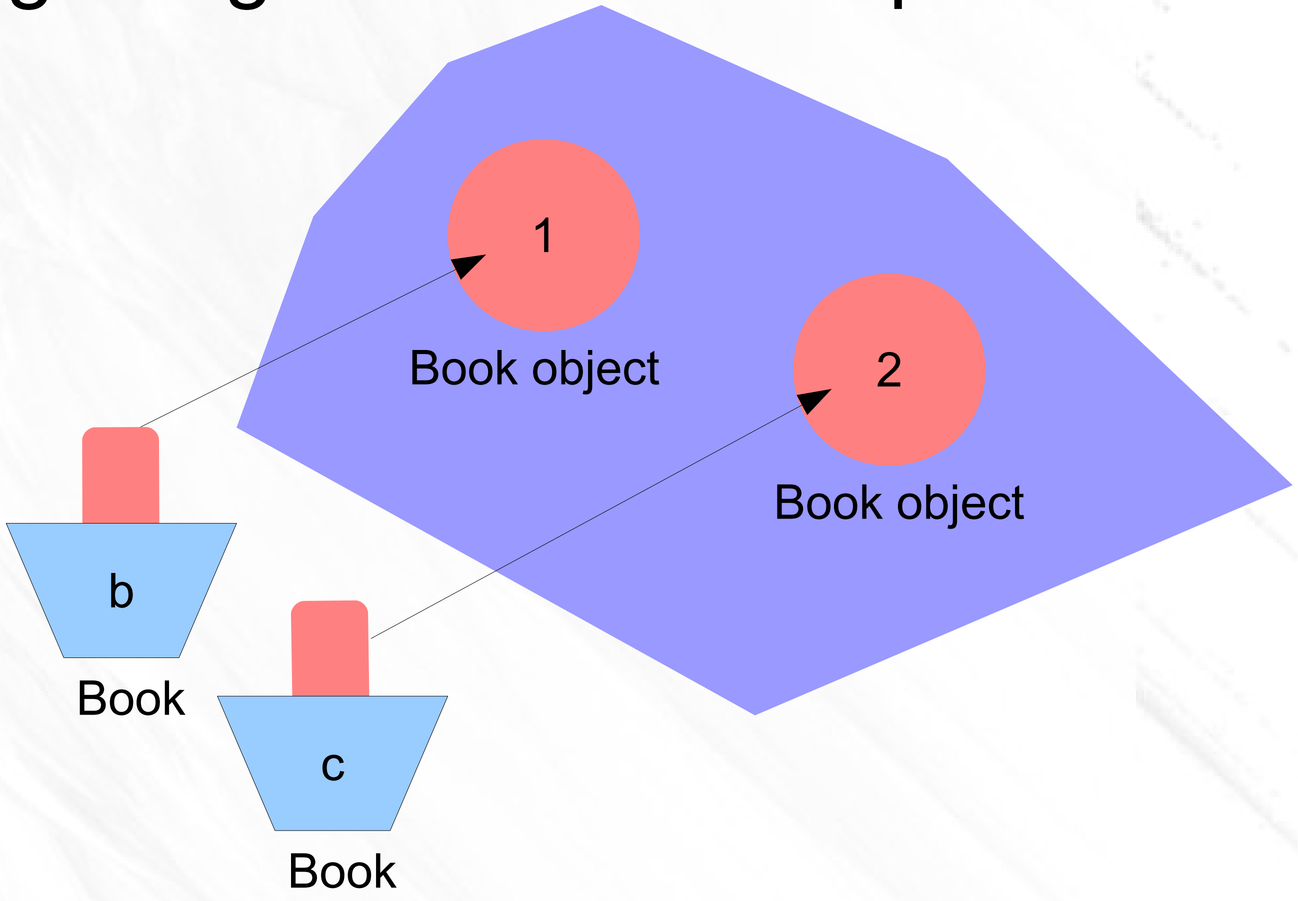
```
Book b = new Book ( );

Book c = new Book ( );
```

Declare two Book reference variables.
Create two new Book objects to the reference variables.

The two Book objects are now living on the heap.

References: 2

Objects: 2



Object refrences

Book d = c;

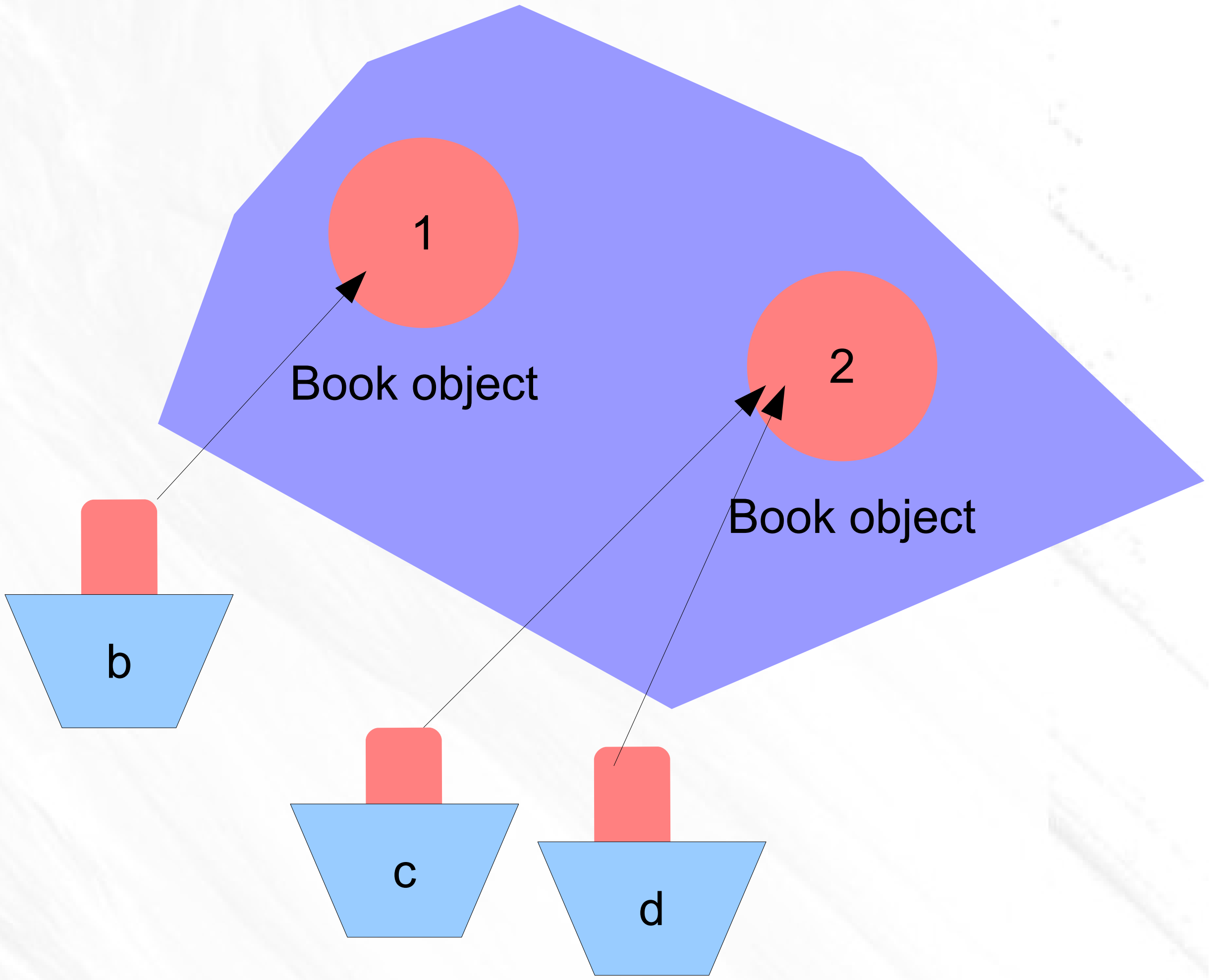
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable ***c*** to variable ***d***. But what does this mean? It's like saying, "Take the bits in ***c***, make a copy of them, and stick that copy into ***d***."

Both *c* and *d* refer to the same object.

The *c* and *d* variables hold two different copies of the same value. Two remotes programmed to one TV.

References: 3

Objects: 2



Object references

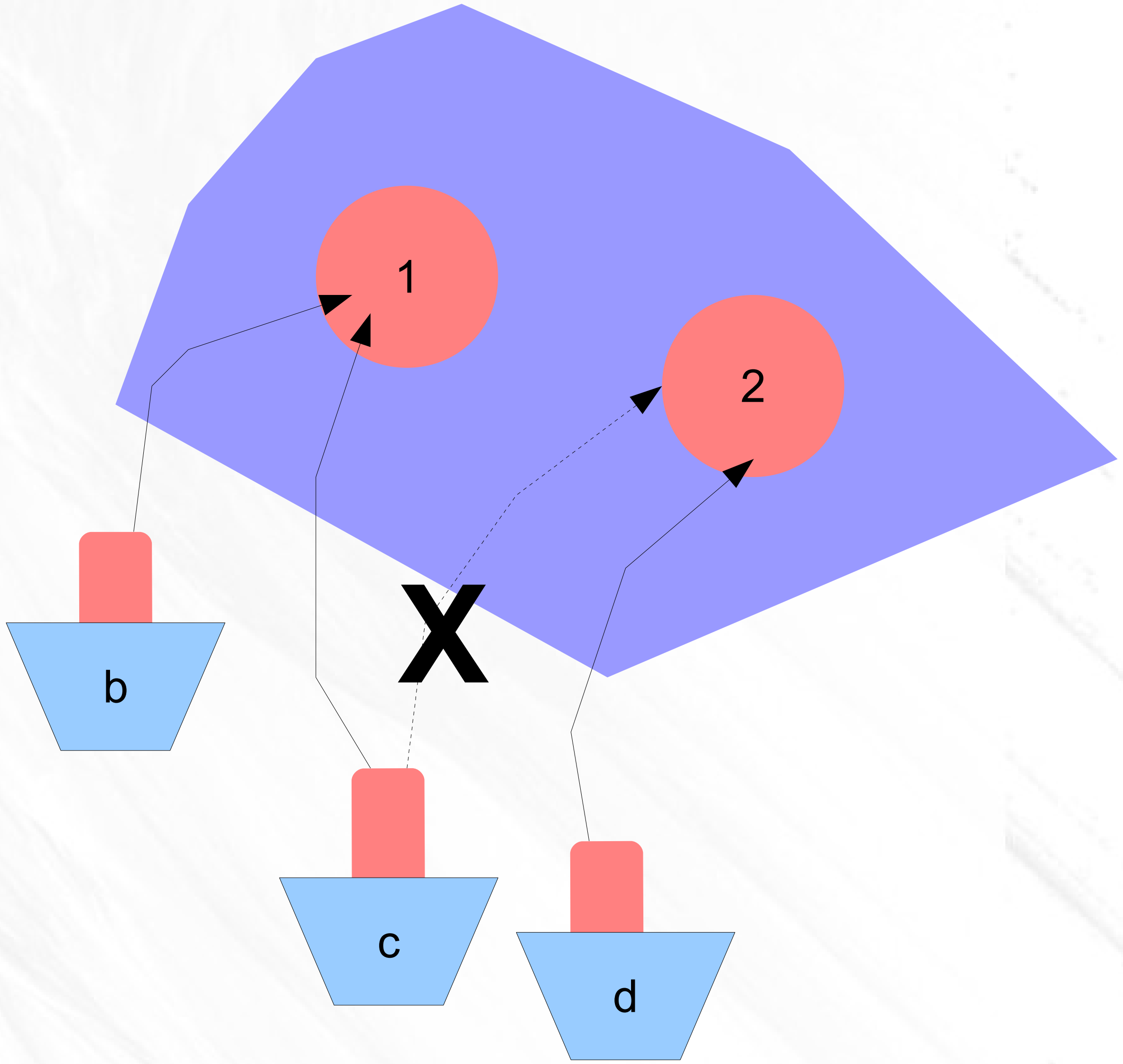
c = b;

Assign the value of variable ***b*** to variable ***c***. By now you know what this means. The bits inside variable ***b*** are copied, and that new copy is stuffed into variable ***c***.

Both b and c refer to the same object.

References: 3

Objects: 2



Object references

Life and death on the heap

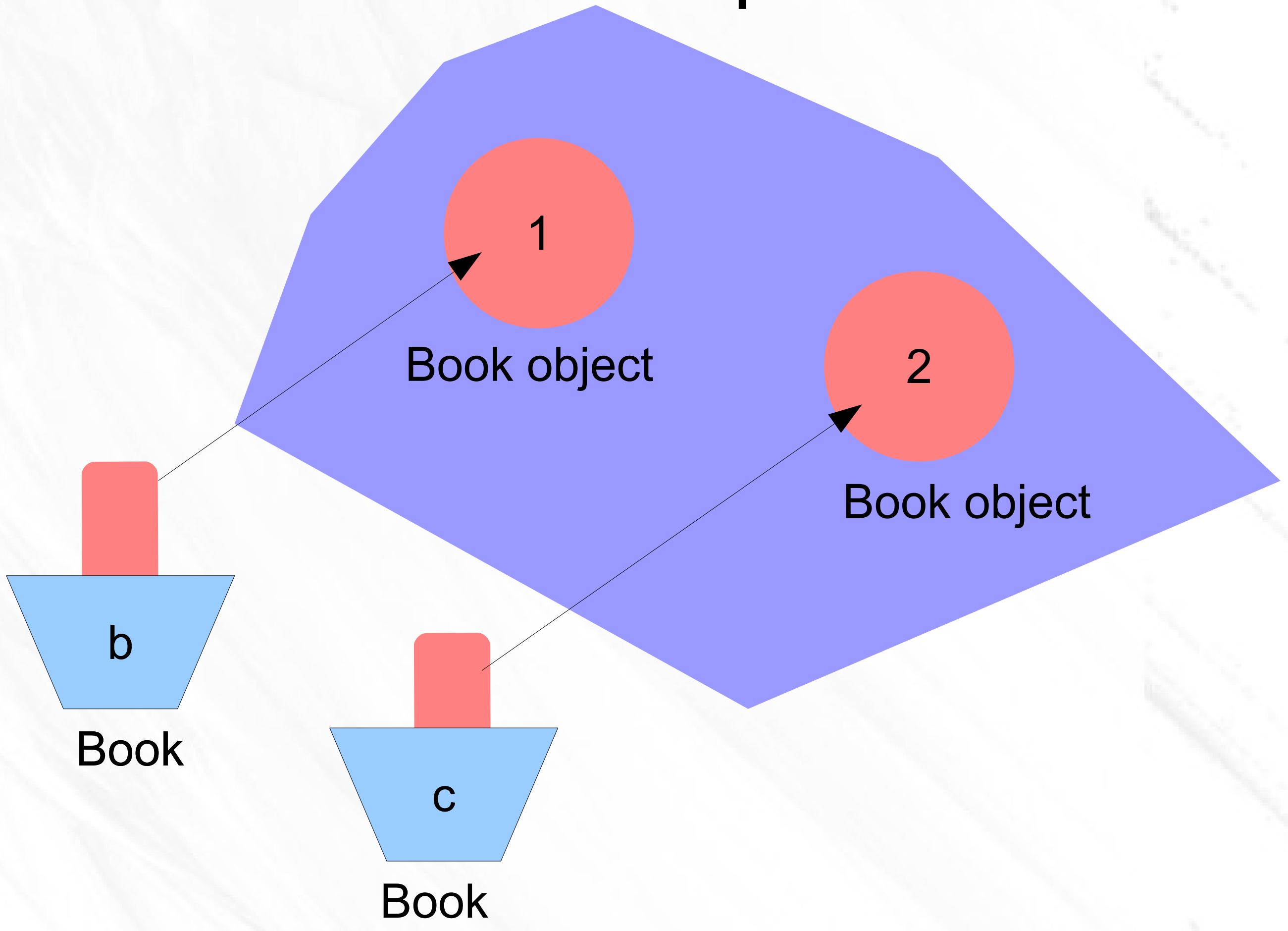
```
Book b = new Book( );

Book c = new Book( );
```

Decalare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two book objects are now living on the heap.

Active References: 2

Reachable Objects: 2



Object references

b = c;

Assign the value of variable ***c*** to variable ***b***. The bits inside variable ***c*** are copied, and that new copy is stuffed into variable ***b***. Both variables hold identical values.

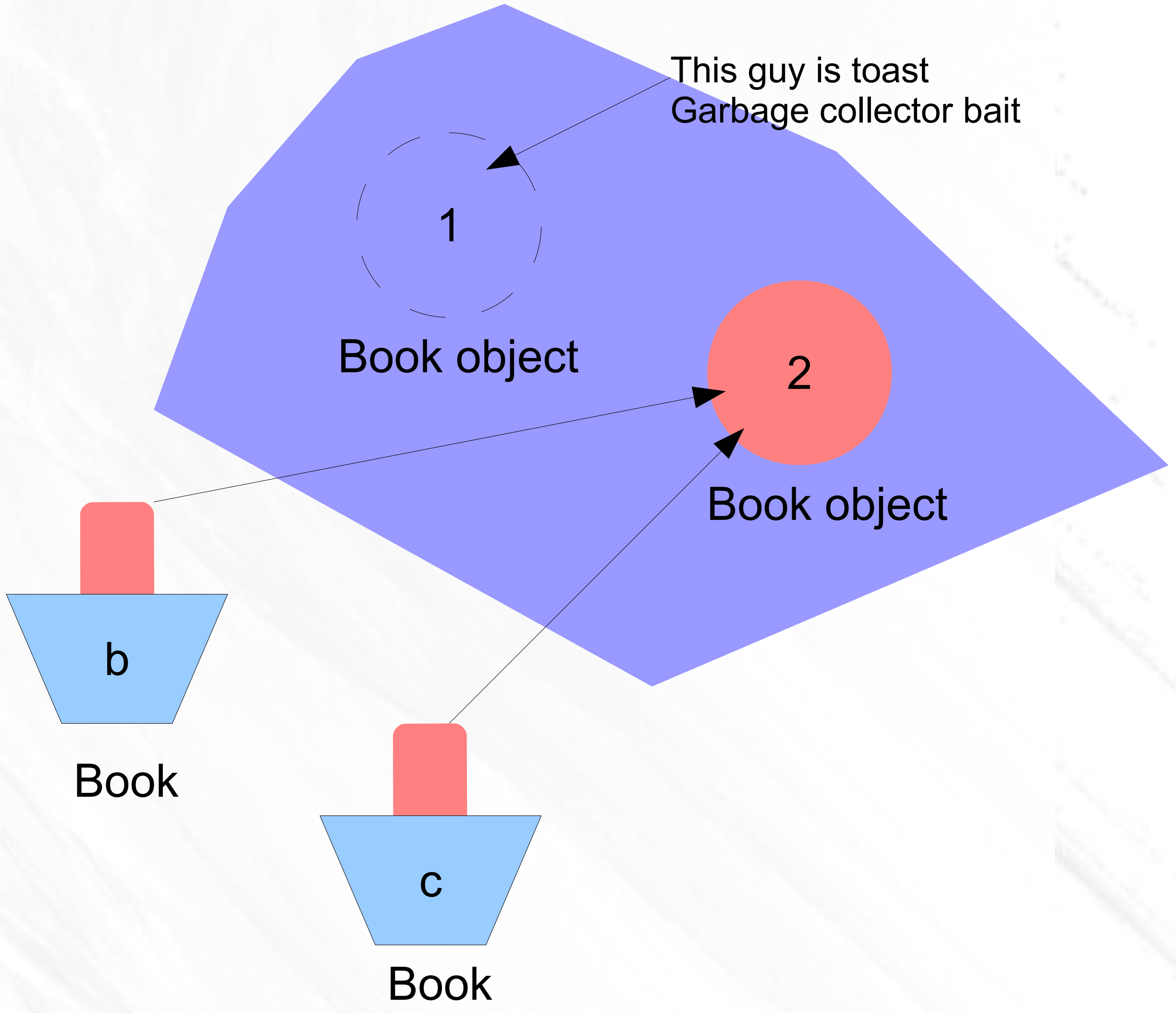
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

The firest object that ***b*** referenced, Object , has no more references. *It's unreachable*.



Object refrences

c = null;

Assign the value **null** to variable **c**. This makes **c** a *null reference*, meaning it doesn't refer to anything. But it's still a reference variable, and another Book object can still be assigned to it.

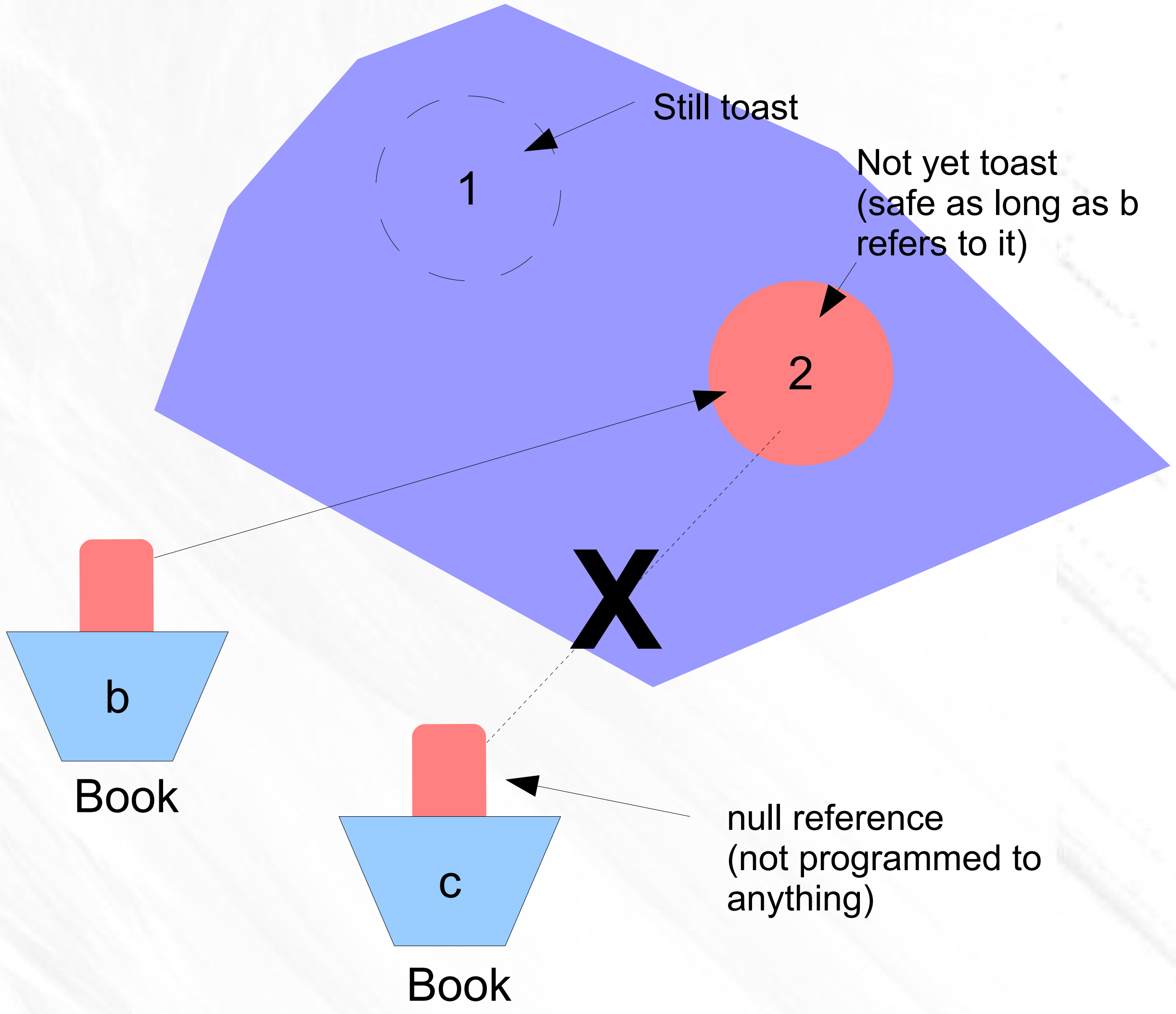
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

null References: 1

Reachable Objects: 1

Abandoned Objects: 1



Object references

Arrays are objects too

- Use arrays when you want a quick, ordered, efficient list of things.
- Arrays give you fast random access by letting you use an index position to get to any element in the array.
- Every element in an array is just a variable (one of the eight primitive types or a reference variable)

Object references

Arrays are objects too

- Anything you would put in a *variable* of that type can be assigned to an *array element* of that type.
  - In an array of type int (int [ ]), each element can hold an int.
  - In an array of type Dog (Dog [ ]) each element can hold a reference (*remote control*) to a Dog.
- **Arrays are always objects, whether they're decalred to hold primitives or object references.**
  - You can have an array object that's declared to *hold* primitive values. The array object can have *elements* which are primitives, but the array itself is *never* a primitive.
- **Regardless of what the array holds, the array itself is always an object!**

Object references

An array is like a tray of cups

1. Declare an int array variable. An array variable is a remote control to an array object.

```
int [ ] nums;
```

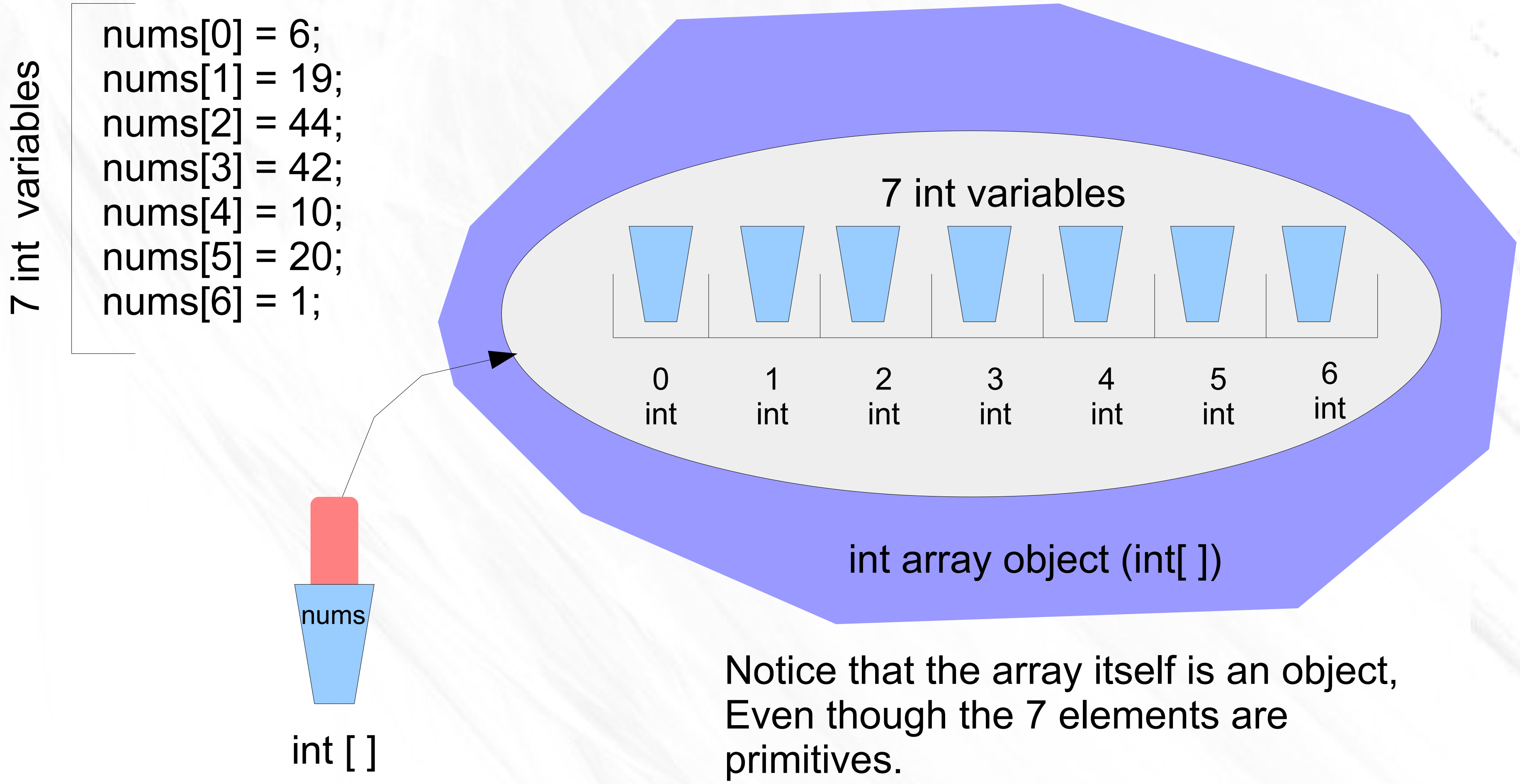
2. Create a new int array with a length of 7, and assign it to the previously-declared int [ ] variable nums

```
nums = new int[7];
```

3. Give each element in the array an int value. Remember, elements in an int array are just int variables.

Object references primitive arrays

```
nums[0] = 6;
nums[1] = 19;
nums[2] = 44;
nums[3] = 42;
nums[4] = 10;
nums[5] = 20;
nums[6] = 1;
```

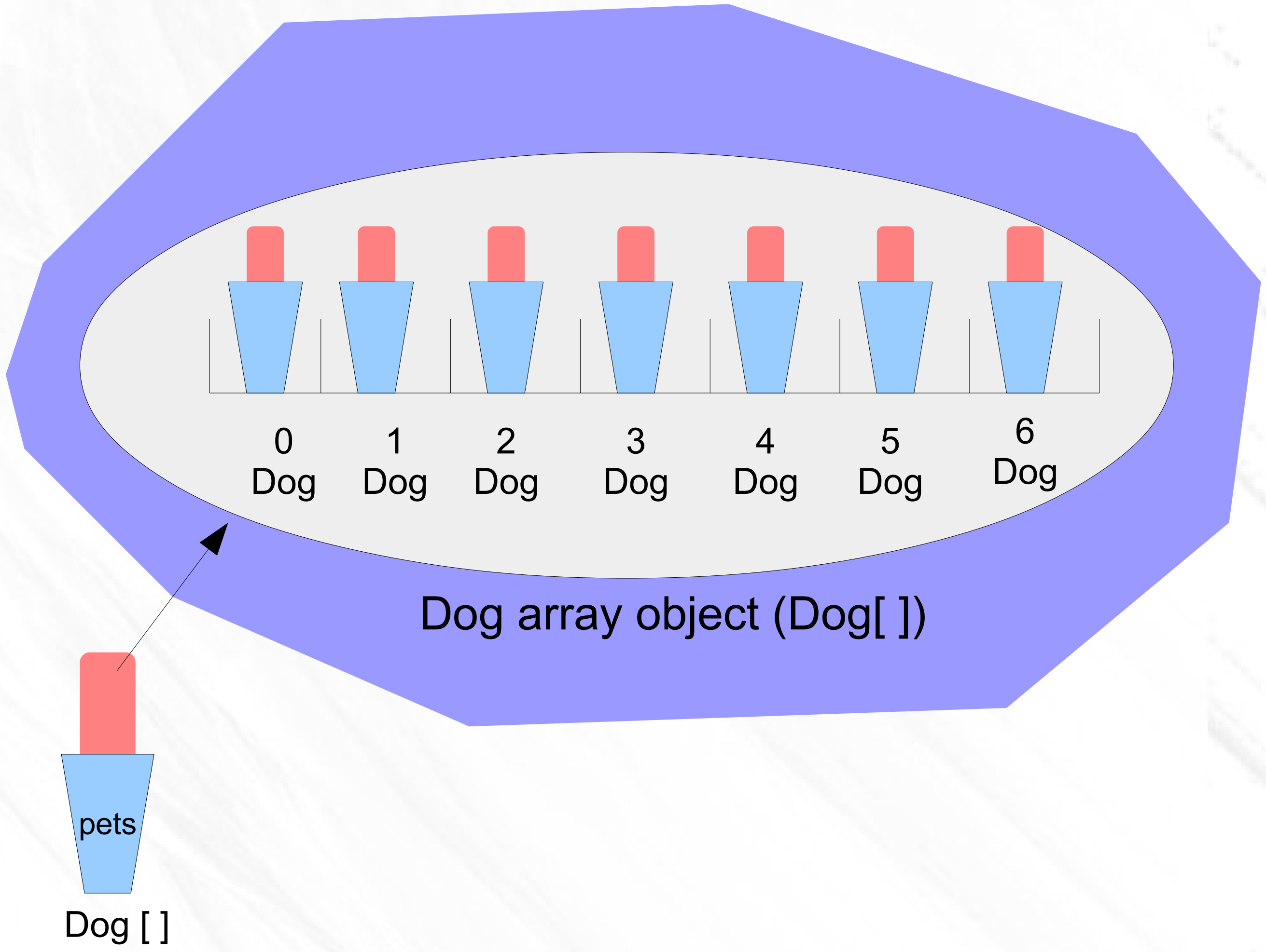


Notice that the array itself is an object,
Even though the 7 elements are
primitives.

Object references object arrays

1. Declare a Dog array variable
   Dog [ ] d pets;

2. Create a new Dog array with a length of 7, and assign it to the previously-declared **Dog [ ]** variable **pets**
   pets = new Dog[7];

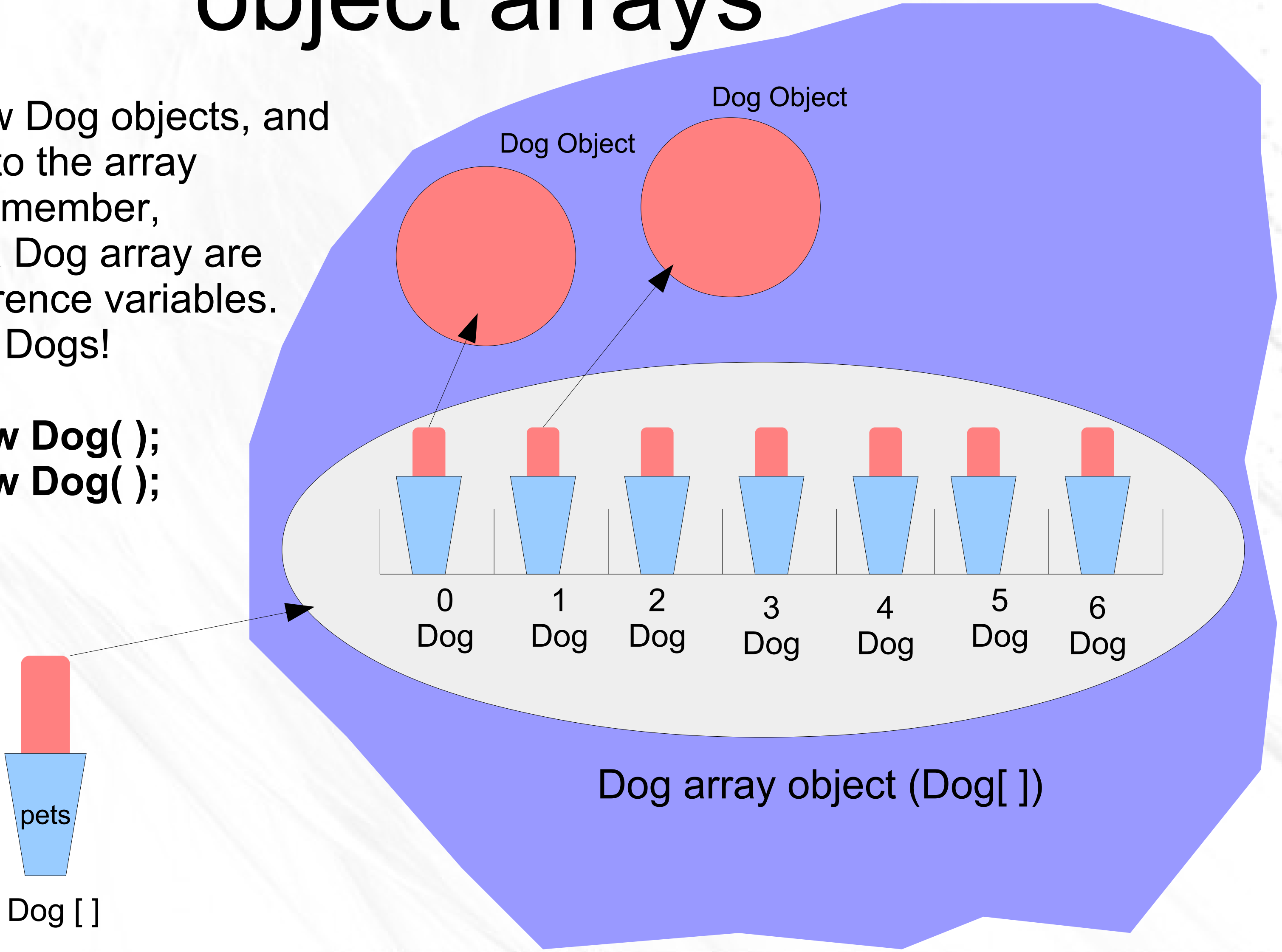


What's missing?
Dogs! We have an array of Dog *references*, but no actual Dog *objects*!

Object references object arrays

3. Create new Dog objects, and assign them to the array elements. Remember, elements in a Dog array are just Dog reference variables. We still need Dogs!

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



Object references object arrays

Remember: Java cares about type!

Once you've declared an array, you can't put anything in it except things that are of the declared array type.

For example:

You can't put a Cat into a Dog array

You can't stick a **double** into an **int** array (spillage)

However:

you can put a **byte** into an **int** array, because a **byte** will always fit into an **int**-sized cup.

This is known as **implicit widening**.

Object references controlling objects

Control your Dog

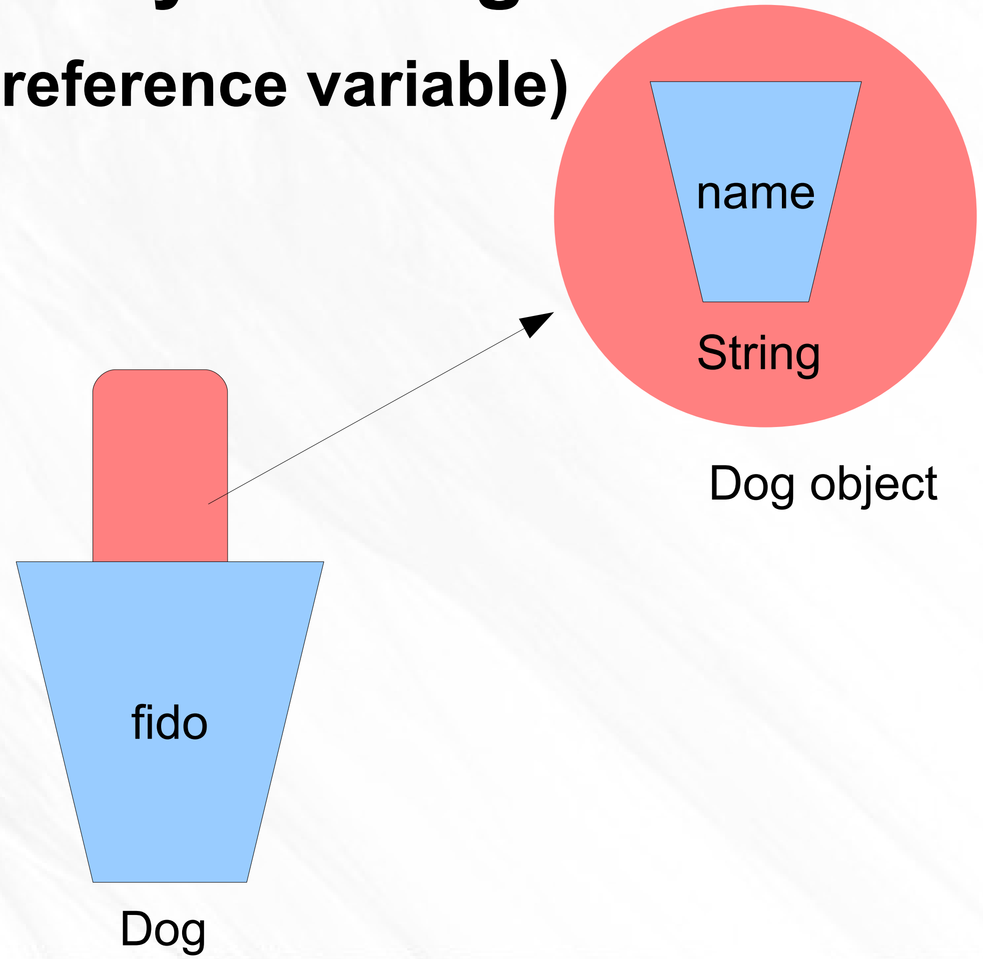
(with a reference variable)

```
Dog fido = new Dog ( );
fido.name = “Fido”;
```

We created a Dog object and used the dot operator on the reference variable ***fido*** to access the name variable

We can use the ***fido*** reference to get the dog to bark( ) or eat( ) or chaseCat( ).

```
fido.bark( );
fido.chaseCat( );
```



Object references controlling objects

What happens if the Dog is in a Dog array?

We know we can access the Dog's instance variables and methods using the dot operator, but *on what?*

When the Dog is in an array, we don't have an actual variable name (like ***fido***). Instead we use array notation and push the remote control buttton (dot operator) on an object at a particular index (postion) in the array:

Dog [ ] myDogs = new Dog[3];
myDogs[0] = new Dog( );
myDogs[0].name = “Fido”;
myDogs[0].bark( );

```
Cclass Dog {
        String name;
        Ppublic static void main (String [ ] args) {
        //make a Dog object and access it
        Dog dog1 = new Dog( );
        dog1.bark( );
        dog1.name = “Bart”;

        //now make a Dog array
        Dog [ ] myDogs = new Dog [3];
        //and put some dogs in it
        myDogs[0] = new Dog( );
        myDogs[1] = new Dog( );
        myDogs[2] = dog1;

        //now access the Dogs using the array
        //references
        myDogs[0].name = “Fred”;
        myDogs[1].name = “Marge”;

        //Hmmm... what is myDogs[2] name?
        System.out.print(“last dog's name is “ + myDogs[2].name);

        //Now loop through the array and tell all dogs to bark
        int x = 0;
        while(x<myDogs.length) {
                myDogs[x].bark( );
                x = x + 1;
                }
        }

        public void bark( ) {
                System.out.println( name + “ says Ruff!”);
        }
        public void eat( ) { }
        public void chaseCat( ) { }
}
```

Take-aways

- Variables come in two categories: primitive and reference
- Variables must always be declared with a name and a type
- A primitive variable value is the bits representing the value(5, 'a', true, 3.1416, etc.).
- A reference variable value is the bits representing a way to get to an object on the heap.
- A reference variable is like a remote control. Using the dot operator (.) on a reference variable is like pressing a button on the remote control to access a method or instance variable.
- A reference variable has a value of **null** when it is not referencing any object
- An array is always an object, even if the array is declared to hold primitives. There is no such thin as a primitive array, only an array that holds primitives.