Section 4

Primitives & References:
Know Your Variables

Learning Outcomes

- After this lecture you should be able to:
  - Distinguish between instance variables & local variables
  - Outline the differences between primitive & reference variables
  - Declare, create & assign objects
  - Describe the purpose of Constructors
  - Outline Encapsulation and the use of getter/setter methods
  - Describe the life & death of an object on the GC heap
  - Describe how arrays work
  - Create arrays of primitive variables and reference variables
  - ***Reading*** and ***studying*** recommended text is essential to improve your understanding of the above

Primitives & References

- Variables thus far
  - Object state (instance variables - values are unique to each *instance* of a class)
  - Local variables (declared within a method)
- Variables can also be used as
  - Arguments (values sent to methods)
  - Return types (sent back to method caller)
- Variables come as
  - Primitive types (byte, short, int, long, float, double, boolean, and char)
    - compiler will assign a reasonable default value for fields
    - However, for local variables, a default value is never assigned
  - Object Reference (Strings, Arrays, references to objects)

Variables

- Variables
  - Java is statically-typed
    - all variables must be declared before they can be used
  - Different types, different sizes
  - type determines the values it may contain
- Variables must have
  - a type (String, int …)
  - a name (title, rating …)

```
String title;
int rating;
```

- An object of type x
  - type and class are synonyms

```
Bicycle bike1
Dog spot
```

Primitive declarations with assignments:

```
int x;
x = 234;
byte b = 89;
boolean isFun = true;
double d = 3456.98;
char c = 'f';
int z = x;
boolean isPunkRock;
isPunkRock = false;
boolean powerOn;
powerOn = isFun;
long big = 3456789;
float f = 32.5f;
```

Note the 'f'. Gotta have that with a float, because Java thinks anything with a floating point is a double, unless you use 'f'.

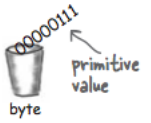
Name your Variables

- Variable names are
  - case-sensitive
  - convention - begin your variable names with a letter, not "$" or "_"
  - subsequent characters may be letters, digits, dollar signs, or underscore characters
  - choosing a name for your variables, use full words instead of cryptic abbreviations
  - cannot use reserved words

Back to Objects

- Object reference
  - No such thing as an object variable
  - Only an object reference variable
  - An object reference variable holds bits that represent a way to access an object
  - It doesn't hold the object itself, it holds an address/pointer
- Primitive variables
  - contain bits representing actual values
  - Bits representing 7 go into variable

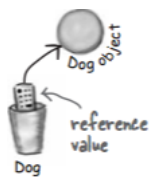
```
byte x = 7;
```



- Object references
  - contain bits representing a way to get to an object
  - Use the dot (.) operator

```
Dog spot = new Dog ();
spot.size = 20;
```

  - Dog object itself doesn't go into the variable



Declare, Create & Assign

- Declare a reference variable
  - JVM allocates space for a reference variable and names it *myBike*

```
Bicycle myBike
```

  - The reference variable is of type Bicycle
- Create an object
  - JVM allocates space for a new Bicycle on the heap

```
new Bicycle( );
```

- Link the object and the reference
  - Assigns the new Bicycle to the reference variable

```
Bicycle myBike = new Bicycle( );
```

Constructor

- A constructor is a piece of code that runs when someone says *new* on a class type
- Must have the same name as the class and NO return type
- If you don't put a constructor in your class the compiler will (default is a no-argument constructor)

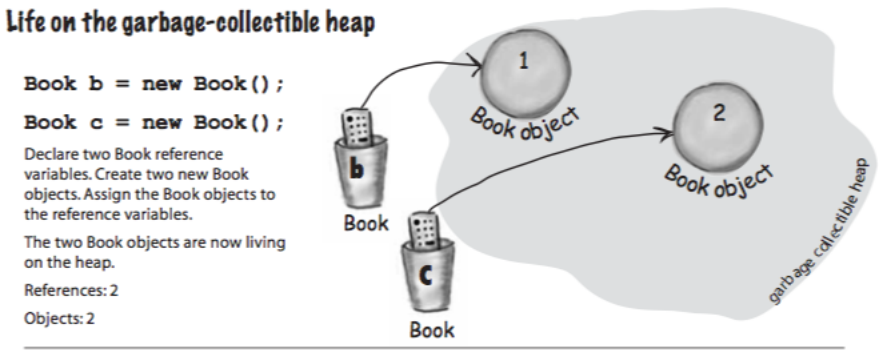
```
public Book(){ }
```

- Good practice to provide a constructor

Encapsulation

- Good OO practice to *hide* your data
- Remember Methods can take in values and return values
- Mark your instance variables *private* and provide public *getters* and *setters* for access control
- More to follow...

Life on the GC heap



Life on the GC heap

```
Book d = c;
```

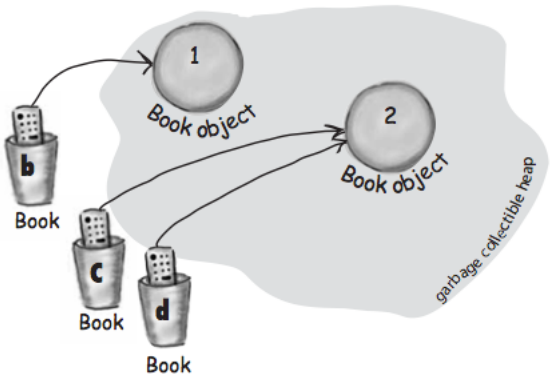
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable ***c*** to variable ***d***. But what does this mean? It's like saying, "Take the bits in ***c***, make a copy of them, and stick that copy into ***d***."

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



Life on the GC heap

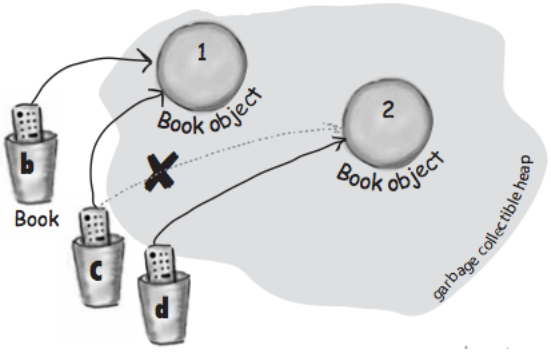
```
c = b;
```

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



Book Class

```
public class Book {

    // instance variable (private)
    private String title;

    // constructor
    public Book(){
        title = "unassigned";
    }

    // methods (public getters & setter)
    public String getTitle() {
        return title;
    }

    public void setTitle(String t){
        title = t;
    }

}
```

Book Tester

```
public class BookTester {

    public static void main(String[] args) {

        Book b = new Book();
        Book c = new Book();
        Book d = c;

        b.setTitle("Head First Java ");
        c.setTitle("Thinking in Java");

        System.out.println("Title of b is " + b.getTitle());
        System.out.println("Title of c is " + c.getTitle());
        System.out.println("Title of d is " + d.getTitle());

        c = b;
        System.out.println("Title of c is now " + c.getTitle());

    }

}
```

Death on the GC heap

Life and death on the heap

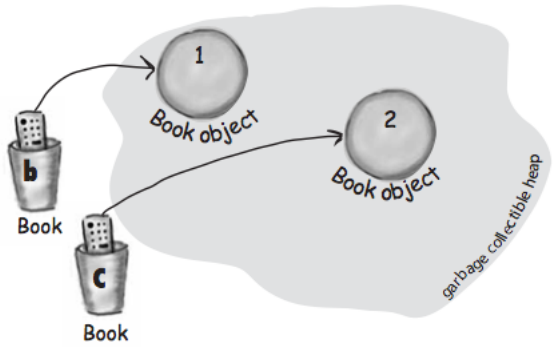
```
Book b = new Book();
Book c = new Book();
```

Declare 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



Death on the GC heap

```
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.

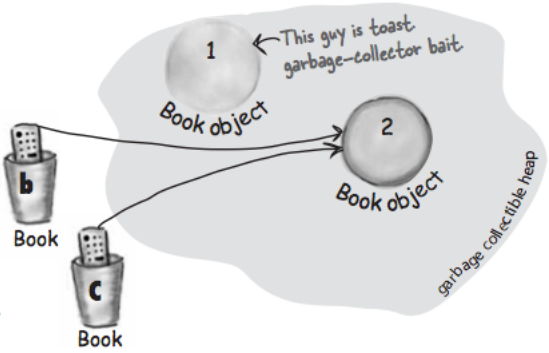
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

The first object that ***b*** referenced, Object 1, has no more references. It's *unreachable*.



Objects on the GC heap

`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.

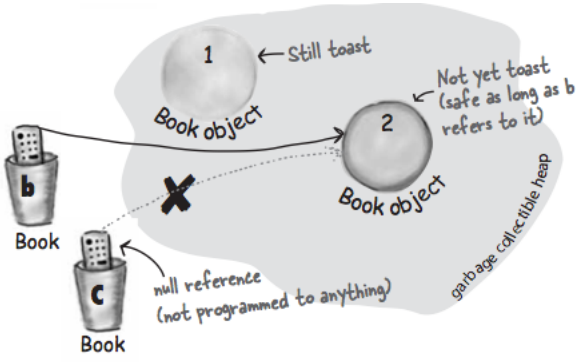
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

null References: 1

Reachable Objects: 1

Abandoned Objects: 1



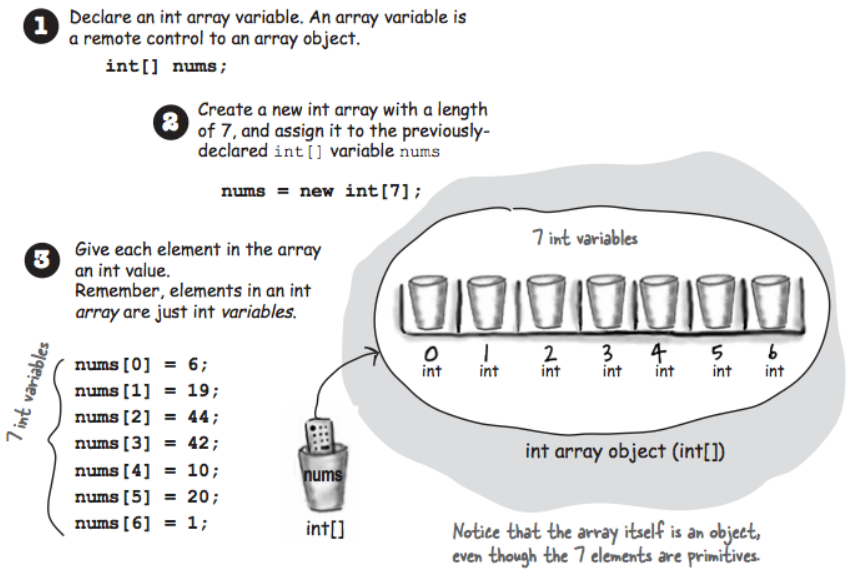
Book Tester

```
public class BookTester {

    public static void main(String[] args) {

        Book b = new Book();
        Book c = new Book();

        b.setTitle("Head First Java ");
        c.setTitle("Thinking in Java");

        System.out.println("Title of b is " + b.getTitle());
        System.out.println("Title of c is " + c.getTitle());

        b = c;
        System.out.println("Title of b is now " + b.getTitle());

        c = null;
        System.out.println("Title of c is " + c.getTitle());
        // Throws a java.lang.NullPointerException
    }

}
```

Arrays

- Arrays are a collection of items of the same type
- Allow fast random access by using an index position to get at any element in the array
- Anything you can put in a variable of that type can be assigned to an array element of that type
  - An array of type double (`double[ ];`), each element can hold a double
  - An array of type Bicycle (`Bicycle [ ];`), each element can hold a Bicycle?
  - NO it can hold a reference to a Bicycle object, not the object itself
- An array is an object – even if it's an array of primitives

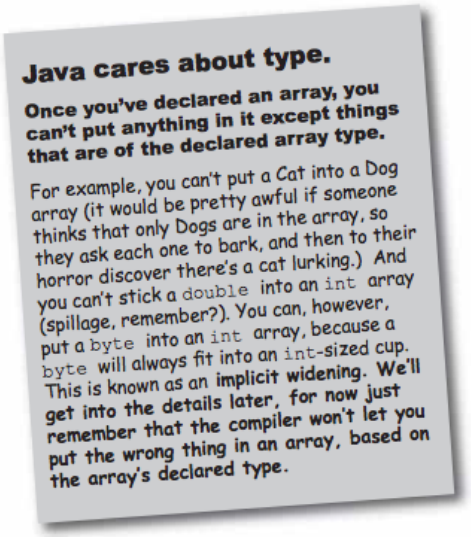
Arrays



Array Code

```
public class Array {

    public static void main(String[] args) {

        int[] nums; // declare array variable

        // create an int array and assign it to the int[] variable nums
        nums = new int[7];

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

        for(int i=0; i< nums.length; i++){
            System.out.println("nums[" + i + "] = " + nums[i]);
        } // end loop

    } // end method

} // end class
```

Arrays & Types



Dog Class - Revisited

```
public class Dog {

    String name;

    public void bark() {

        System.out.println(name + " says Ruff!");

    }

    public void eat() { }

    public void chaseCat() { }

}
```

```
public class DogTester {

    public static void main(String[] args) {

        Dog dog1 = new Dog();
        dog1.name = "Bart";
        dog1.bark();

        Dog[] myDogs = new Dog[3];
        myDogs[0] = new Dog();
        myDogs[1] = new Dog();
        myDogs[2] = dog1;

        myDogs[0].name = "Fred";
        myDogs[1].name = "Marge";

        System.out.print("last dog's name is ");
        System.out.println(myDogs[2].name);

        int x = 0;
        while (x < myDogs.length) {
            myDogs[x].bark();
            x = x+1;
        }
    }
}
```

Summary

- Variables - Primitive & Reference
- Declare & Create Reference
- Constructors
- Encapsulation – an introduction
- public vs private
- Life & Death on the Heap
- Arrays
- Arrays & Types