Chapter 3

Know your variables

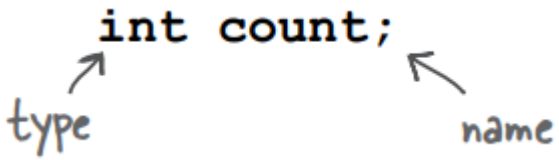
- So far we use variables in two places – as the instance variables and as the local variables
- Of course, you know that variables can be used for arguments and return types
- In this chapter, we will unwrap Java types and look at what you can **declare** as a variable, what you can **put** in a variable, and what you can **do** with a variable

- Java cares about type. it won't let you put a floating point number into an integer variable, unless you **acknowledge to the compiler** that you know you might lose precision
- The compiler can spot most problems such as

```
Rabbit hopper = new Giraffe();
```

- You must declare the type of your variable. Is it an integer? a Dog? A single character?
- Variables come in two flavors: primitive and object reference
- Primitives hold fundamental values including integers, booleans, and floating point numbers
- Object references hold, well, references to objects

- The declaration rules:
- Variables must have a **type** and a **name**



- We declare a variable named **count** as type **integer**
- When you think of Java primitive variables, think of cups.
- A primitive variable is just a cup. A container. It **holds** something.

- Primitives are like the cups they have at the coffeehouse.
- If you've been to a Starbucks, you know they come in different sizes, and each has a name like 'short', 'tall', and, "I'd like a grande mocha decaff with extra whipped cream"
- In Java, primitives come in different **sizes**, and those sizes have names. When you declare a variable in Java, you must declare it with a specific type.

- There are 8 primitive types in Java:

Primitive Types

Type	Bit Depth	Value Range
boolean and char		
boolean	(JVM-specific)	***true*** or ***false***
char	16 bits	0 to 65535
numeric (all are signed)		
integer		
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-huge to huge
floating point		
float	32 bits	varies
double	64 bits	varies

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'.

Question: we have **char** in primitive types, how about **strings**?

- Be sure the value can fit into the variable
- **You can't put a large value into a small cup** (we call it *spillage*)
- The compiler tries to help prevent this if it can tell from your code that something's not going to fit in the container (variable/cup) you're using

```
int x = 24;
byte b = x;
//won't work!!
```

- The value of x is 24, and 24 is definitely small enough to fit into a byte
- All the compiler cares about is that you're trying to put a big thing into a small thing, and there's the possibility of spilling

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

`int size = 32;`	declare an int named *size*, assign it the value *32*
`char initial = 'j';`	declare a char named *initial*, assign it the value *'j'*
`double d = 456.709;`	declare a double named *d*, assign it the value *456.709*
`boolean isCrazy;`	declare a boolean named *isCrazy* (no assignment)
`isCrazy = true;`	assign the value *true* to the previously-declared *isCrazy*
`int y = x + 456;`	declare an int named *y*, assign it the value that is the sum of whatever *x* is now plus *456*

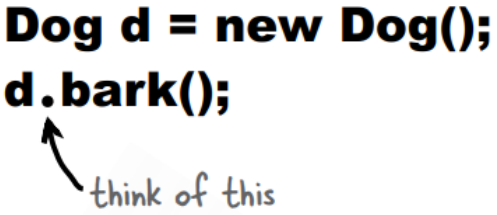
- The compiler won't let you put a value from a large cup into a small one. But what about the other way—pouring a small cup into a big one?
- The answer is yes.
- See if you can figure out which of these are legal and which aren't

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```

- Naming your variables
- It must start with a letter, underscore (_), or dollar sign ($). You can't start a name with a number
- After the first character, you can use numbers as well
- Do not use the reserved words (public, static, void, int, char, boolean, etc.)
- Do not try to memorize these keywords now!
- Fortunately, most IDEs supports **syntax highlighting**

- How about **objects**?
- There is actually no such thing as an object variable
- There's 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, but it holds something like a pointer. Or an address.
- In Java we don't really know what is inside a reference variable. The JVM knows how to use the reference to get to the object

- Objects live in one place and one place only — the **garbage collectible heap**
- A **primitive** variable is representing the **actual value** of the variable
- A **object reference** variable is representing *a way to get to the object*.



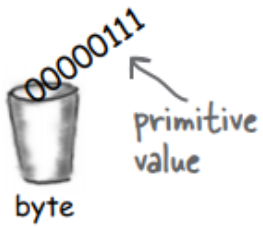
- You use the dot operator (.) on a reference variable to say, “use the object referenced by the variable to invoke the method after the dot”

- An object reference is just another variable value

Primitive Variable

```
byte x = 7;
```

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

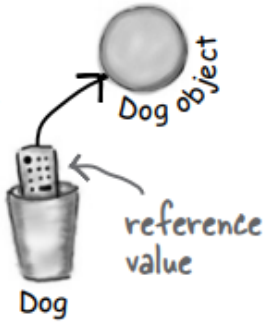


Reference Variable

```
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!



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

1 Declare a reference variable

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

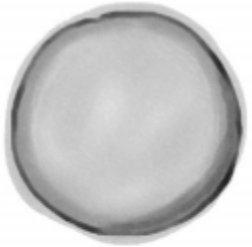
Tells the JVM to allocate space for a reference variable, and names that variable *myDog*. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.

Dog

2 Create an object

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

Tells the JVM to allocate space for a new Dog object on the heap (we'll learn a lot more about that process, especially in chapter 9.)

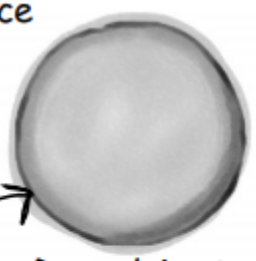


Dog object

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.***



Dog object

Dog

- How big is a reference variable?
- Does that mean that all object references are the same size, regardless of the size of the actual objects to which they refer?
- Can I do arithmetic on a reference variable, such as increment it?
- How to represent a reference variable who refers to *nothing*?
- Can I have more than one reference variables who refer to the same object?
- What happened to a object without any reference variables refer to it?

Life on the garbage-collectible heap

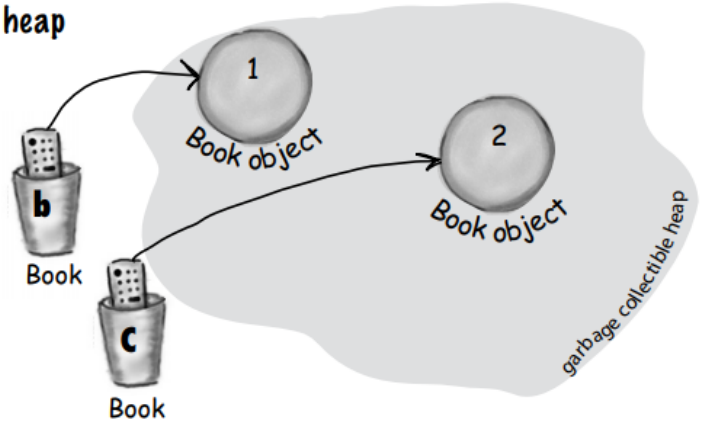
```
Book b = new Book();
Book c = new Book();
```

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.

References: 2

Objects: 2



```
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

```
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

Life and death on the heap

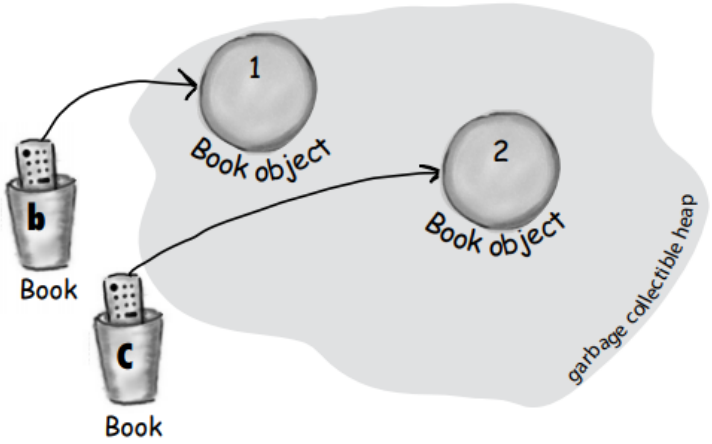
```
Book b = new Book();
Book c = new Book();
```

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




```
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.

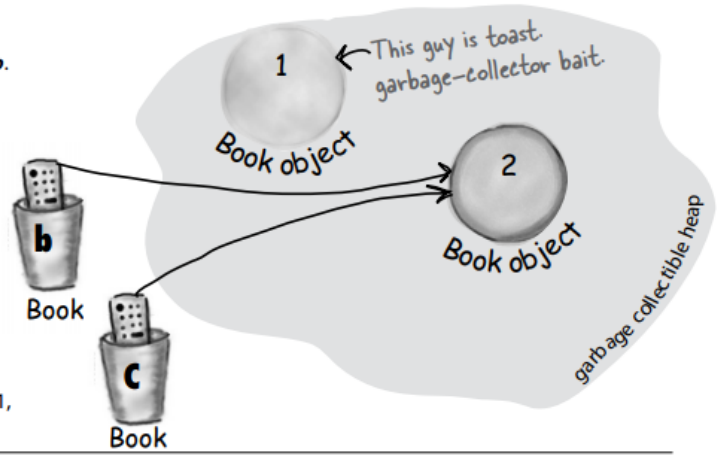
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

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




```
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.

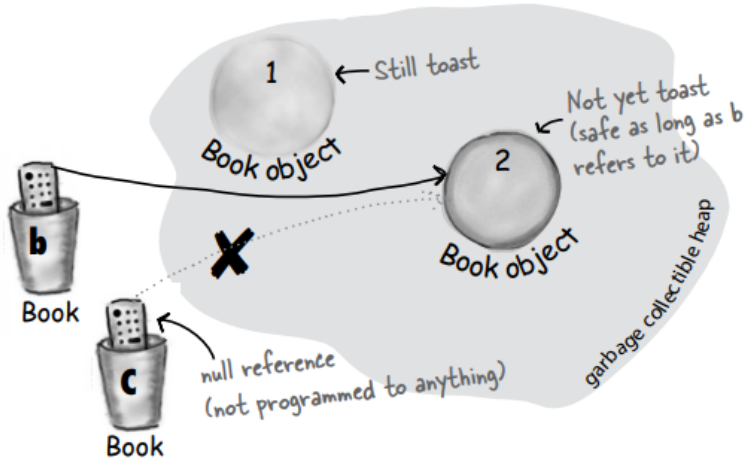
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

null References: 1

Reachable Objects: 1

Abandoned Objects: 1

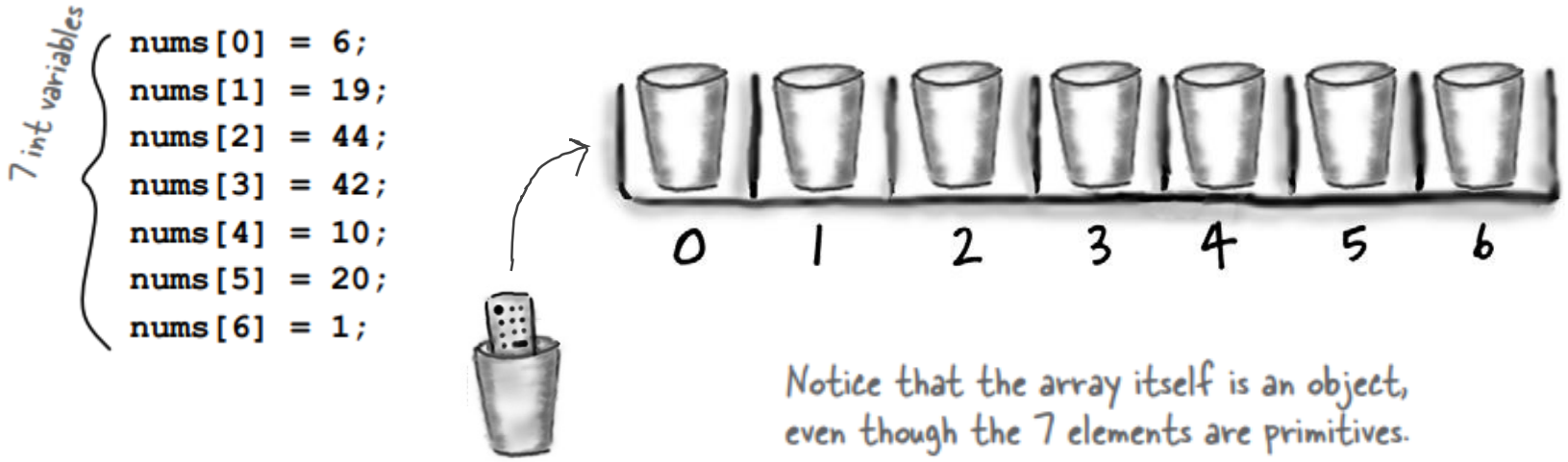


- Array – just like a tray of cups
- Declare an int array variable. An array variable is a reference to an array object

```
int[] nums;
```

- Create a new int array with a length of 7, and assign it to the variable **nums**

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



- Available syntax to declare an array object in Java
- `int[] num;`
- `int[] num = new int [3];`
- `int[] num={1,2,3,4,5};`
- `int num[] = {1,2,3,4,5};`
- `int num[];`
- Invalid syntax
- `int num={1}; // (×)`
- `int num[2]={1,2}; // (×)`
- `Int num[3]; // (×)`

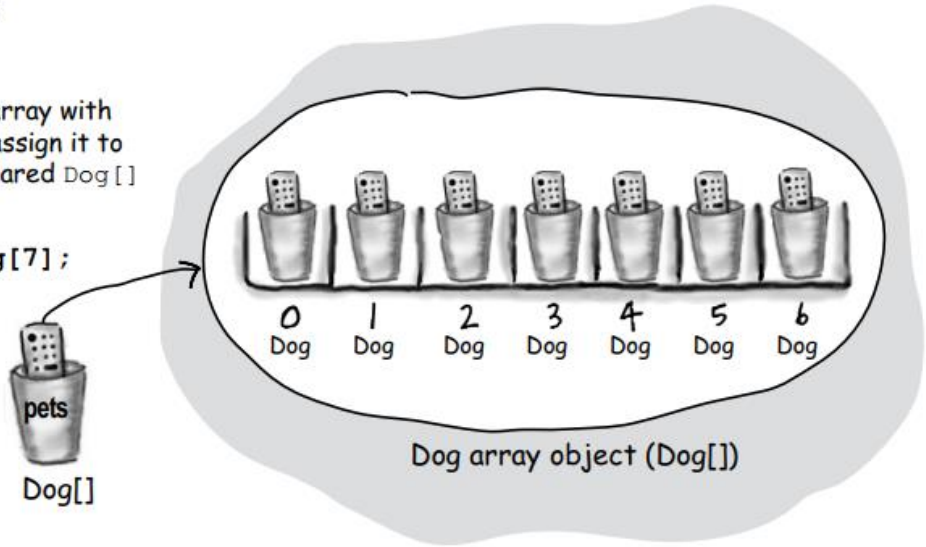
- The Java standard library includes lots of sophisticated data structures including maps, trees, and sets
- However, arrays are great when you just want a quick, ordered, efficient list of things.
- In an array of type int (int[]), each element can hold an int
- How about an array of type Dog (Dog[])? (Dog is a class)
- In a Dog array, each element can hold a **reference** to a Dog
- Arrays are always objects, whether they're declared to hold primitives or object references

1 Declare a Dog array variable

```
Dog[] 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];
```



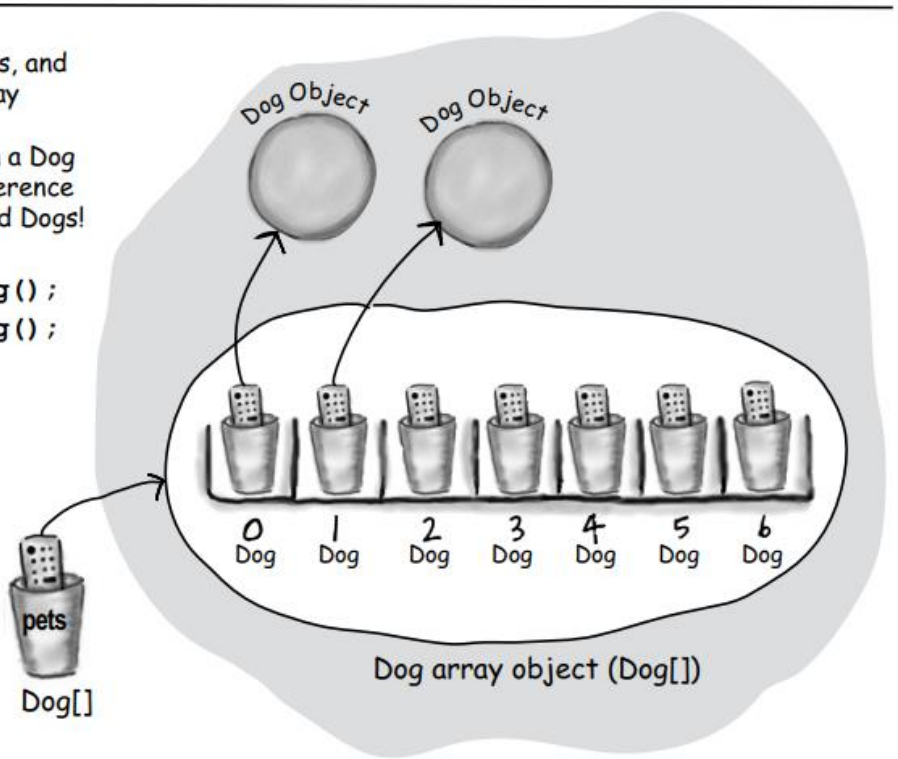
What's missing?

Dogs! We have an array of Dog *references*, but no actual Dog *objects*!

---

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();
```



Sharpen your pencil

What is the current value of pets[2]? _____

What code would make pets[3] refer to one of the two existing Dog objects?

_____

- Quiz: how to create a 2-dimensional (or multi-dimensional array)?

- `int[][] multiArray = new int[4][2];`

- How to **initialize** a two-dimensional array?

```
int[][] myArray = {{1,2,3},{4,5,6}};
```

- Quiz: how many **objects** are created when you write

```
int[][] arr = new int [2][3];
```

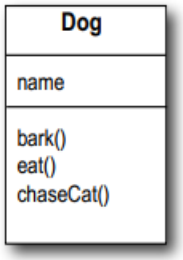
- Quiz
- What is the output?

```
class tmpObj
{
    void doIt(int[] arr)
    {
        arr[0] = 4;
    }
}

public class test1
{
    public static void main(String[] args)
    {
        int[] b = {1,2,3,4,5};
        tmpObj v= new tmpObj();
        v.doIt(b);
        System.out.println(b[0]);
    }

}
```

```
class Dog {
  String name;
  public 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";

    // Hmmmm... what is myDogs[2] name?
    System.out.print("last dog's name is ");
    System.out.println(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() {  }
}
```

arrays have a variable 'length' that gives you the number of elements in the array

A Dog example



Output

```
File Edit Window Help Howl
%java Dog
null says Ruff!
last dog's name is Bart
Fred says Ruff!
Marge says Ruff!
Bart says Ruff!
```

- Summary
- Variables comes in two types: primitive and reference.
- A reference variable represents a way to access an object in the heap.
- A reference variable use a dot operator (.) to invoke methods in an object.
- A null reference variable points to nothing.
- An array is always an object.

- Extension: Java introduces a new var keyword in Java 10
- Instead of doing `String str = "Java"`, you can now just type `var str = "Java"`
- Consider declaring custom class (e.g. Dog), this feature may improve the readability of code

```
IAmASuperLongClass longClassVariable = new IAmASuperLongClass();
→ var longClassVariable = new IAmASuperLongClass();
```

- The var keyword can only be used to declare **local variables**
- The current version of BlueJ (4.2.2) supports Java 11 so you can use this feature if your JVM supports Java 11.
- Quiz: can I write a statement like this?

```
var x;
x = 3;
```

A

```
class Books {
  String title;
  String author;
}

class BooksTestDrive {
  public static void main(String [] args) {

    Books [] myBooks = new Books[3];
    int x = 0;
    myBooks[0].title = "The Grapes of Java";
    myBooks[1].title = "The Java Gatsby";
    myBooks[2].title = "The Java Cookbook";
    myBooks[0].author = "bob";
    myBooks[1].author = "sue";
    myBooks[2].author = "ian";

    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(" by ");
      System.out.println(myBooks[x].author);
      x = x + 1;
    }
  }
}
```

B

```
class Hobbits {

  String name;

  public static void main(String [] args) {

    Hobbits [] h = new Hobbits[3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = "bilbo";
      if (z == 1) {
        h[z].name = "frodo";
      }
      if (z == 2) {
        h[z].name = "sam";
      }
      System.out.print(h[z].name + " is a ");
      System.out.println("good Hobbit name");
    }
  }
}
```

- Extension
- To print out a string, we use the '+' operator to concatenate two strings (variables). What if we try to use the '+' operator on a String object and a primitive type?
- c = "300" + 80;
- Can it compile?

- The Java rules for expression evaluation say that if one operand of the + operator is a String and the other isn't, then the one that isn't is converted to a String, then they're concatenated left to right.