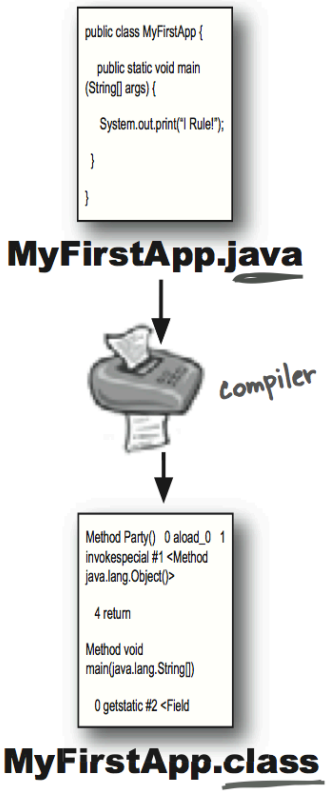
Module 4
Java basics

Running Java application




```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```

1 Save

`MyFirstApp.java`

2 Compile

```
javac MyFirstApp.java
```

3 Run

```
File Edit Window Help Scream
%java MyFirstApp
I Rule!
The World
```

Class description

- The class is declared:

```
<modifier> class <class_name> {
        <data>
        <methods>
}
```

  - `<modifier>` defines class visibility, i.e. classes of which packages can access this class.
  - The **`public`** modifier defines that such a class can be accessed from anywhere.
  - Only one **`public`** class can be defined in a file.

Class description

- The `main()`method is an entry point for any Java program (application).

- The simplest application is composed of one method `main`.

- To run an application in JVM, type in the java command line:

```
java <full_class_name_containing_main>
```

Note! If the class does not have `main` or its signature is wrong, an error `java.lang.NoSuchMethodError` is generated

Method signature

- The `main` method signature.

```
public static void main(String[] args)
```

  - **`void`** – return type.
  - **`args`** – a list of command line arguments.
  - The `main` method should be declared as static.
  - Static method may be (and must be) invoked as:

```
TheClass.staticMethod(...)
```

Note! Program running time equals the `main()` method execution time.

Class example

```
public class Person {
    String name;
    int age;

    int getAge() {
        return age;
    }

    public static void main(String[] args) {
        Person personInstance = new Person();
        personInstance.getAge();
    }
}
```

The classpath concept

- A Java program is a chain of methods invocations of certain object classes. Therefore, during program compilation and runtime the information about the location of byte code of needed classes is required.

- When launching the **`javac`** program that compiles the Java file, you can specify a path list for local file system where dependent classes can be found.

- This specification of paths is called **`classpath`**.

The classpath concept

- Directories in Windows are separated by "`;`", in Unix by "`:`"

- If compiled `Person.java` java file uses some compiled classes located in `c:\lib\classes` it is necessary to indicate the path for compiler through `classpath` flag.

```
javac -d bin
      -sourcepath src
      -classpath C:\lib\classes
      Person.java
```

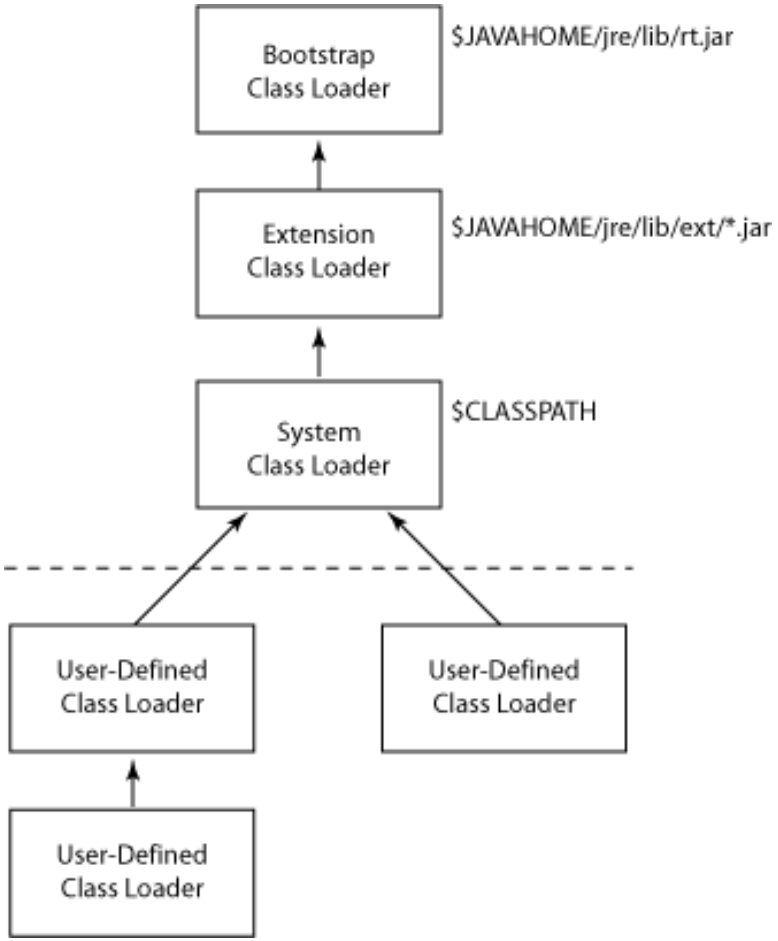
The classpath concept

- Similarly, `classpath` can be specified when launching a Java application. JVM will lookup required classes in specified paths.

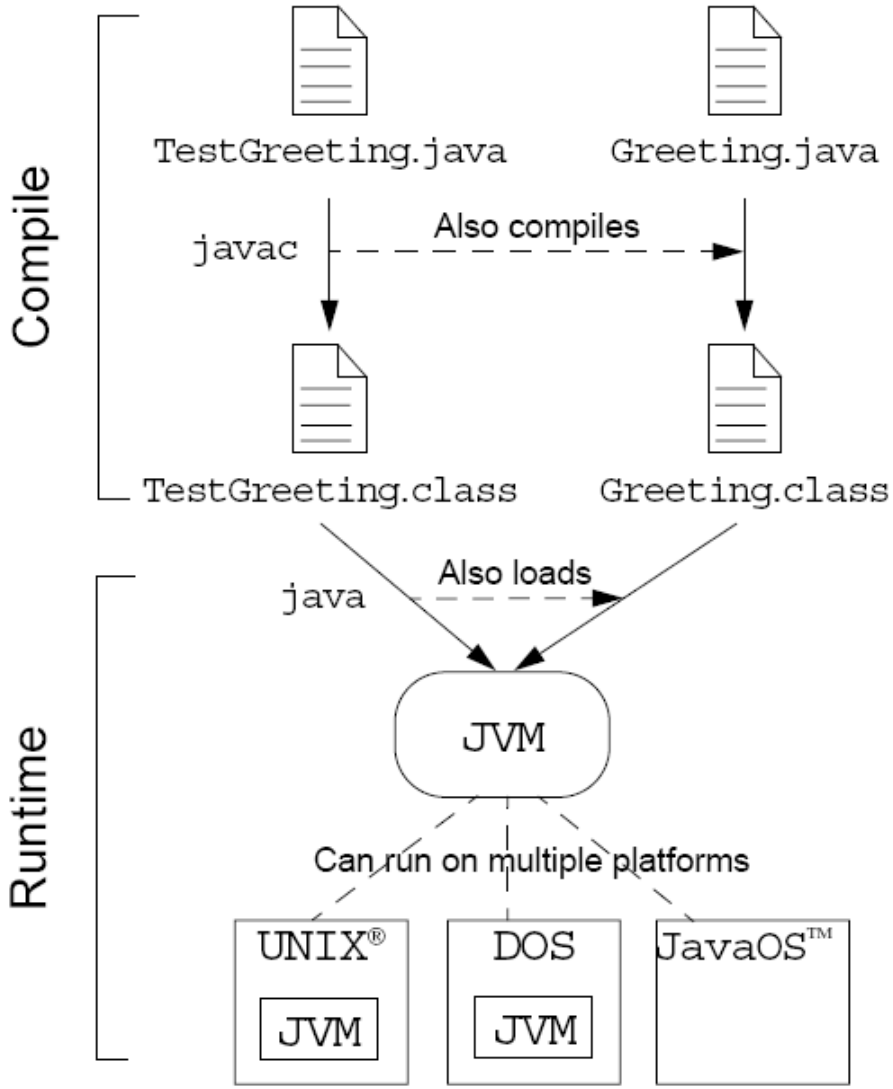
```
java -classpath C:\lib\classes Test
```

Class loading

- JVM has a special loader (bootstrap class loader) that is able to load class byte code from the file system taking into account the full class name.

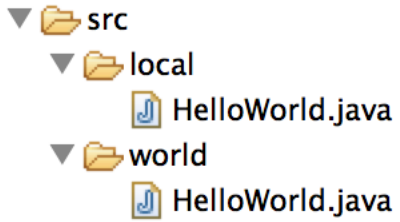


Compiling and running



Packages

- Packages are used to organize classes. Just like two different `readme.txt` files can be located in different directories, java classes can be located in different directories as well.



- Packages allow to avoid collisions when classes have the same name.

Packages

- To place `HelloWorld.java` file into the `world` package it is necessary to specify package name in the file with the help of keyword **`package.`**
- Package declaration should be the first statement.

```
// only comment can be here
package world;

public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello World");
        }
}
```

To execute this class it is necessary to go to the base directory **src** and type: **`$ java –classpath . world.HelloWorld`**

Packages

To use a class located in some package you should specify its **full** name:

```
package another;

public class Test {
  public static void main(String[] args) {
      world.HelloWorld instance = new world.HelloWorld();
  }
}
```

Another possibility is to import class:

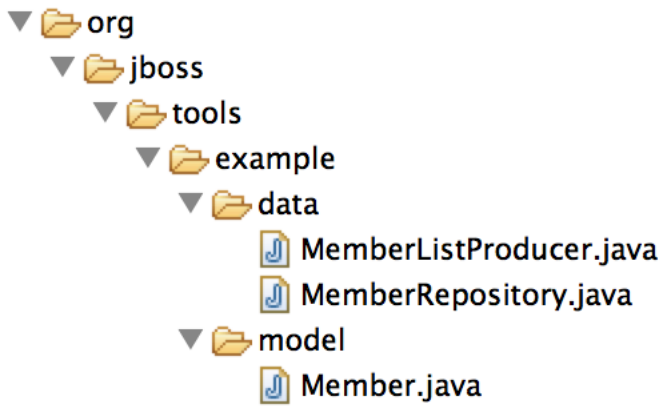
```
package another;
import world.HelloWorld;

public class Test {
  public static void main(String[] args) {
      HelloWorld instance = new HelloWorld();
  }
}
```

Packages

```
package org.jboss.tools.example.data;

public class MemberRepository {
    ...
}
```




```
package org.jboss.tools.example.data;

public class MemberListProducer {
    // MemberRepository can be used without import
    // because it is located in the same package
}
```

```
package org.jboss.tools.example.model;

// other package – class should be imported
import org.jboss.tools.example.data.MemberRepository;

public class Member {
    ...
}
```

Import

- Class imports have nothing to do with loading imported class and doesn’t affect the compilation and running time.

- We can also include all classes of the given package to namespace:

```
import  package.subpackage.*;
```

Jar archives

- There is a way to organize classes in zip archive for more convenient propagation of the **`.class`** file group
- The archive has the `.jar` extension
- If classes are located in a JAR file, it is necessary to specify the archive file name in `classpath`.

```
java -classpath c:\libs\myjar.jar
     thepackage.MainClass
```

Jar archives

- To create Jar file:

```
jar.exe cf myjar.jar MainClass.class
```

c – creates Jar file

f – specifies fine name

Jar archives

- You can also manually define a special manifest file in a JAR file that describes this JAR file.

- The manifest is stored in the META-INF archive directory.

- By default, it is created by the `jar` tool.

Jar archives

- For example, the manifest can indicate the class name that contains `main()`:

```
Manifest-Version: 1.0
Class-Path: ojdbc14.jar
Created-By: Eclipse
Main-Class: my_package.MyClass
```

- In this case the JVM can be launched as follows:

```
java -jar MyJar.jar
```

Exercise

Lab guide:

- Exercise 5

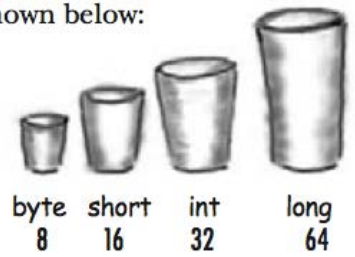
Primitive types

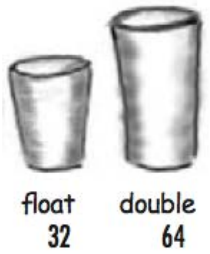
Primitive types

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

shown below:





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;
```

Logic primitive types

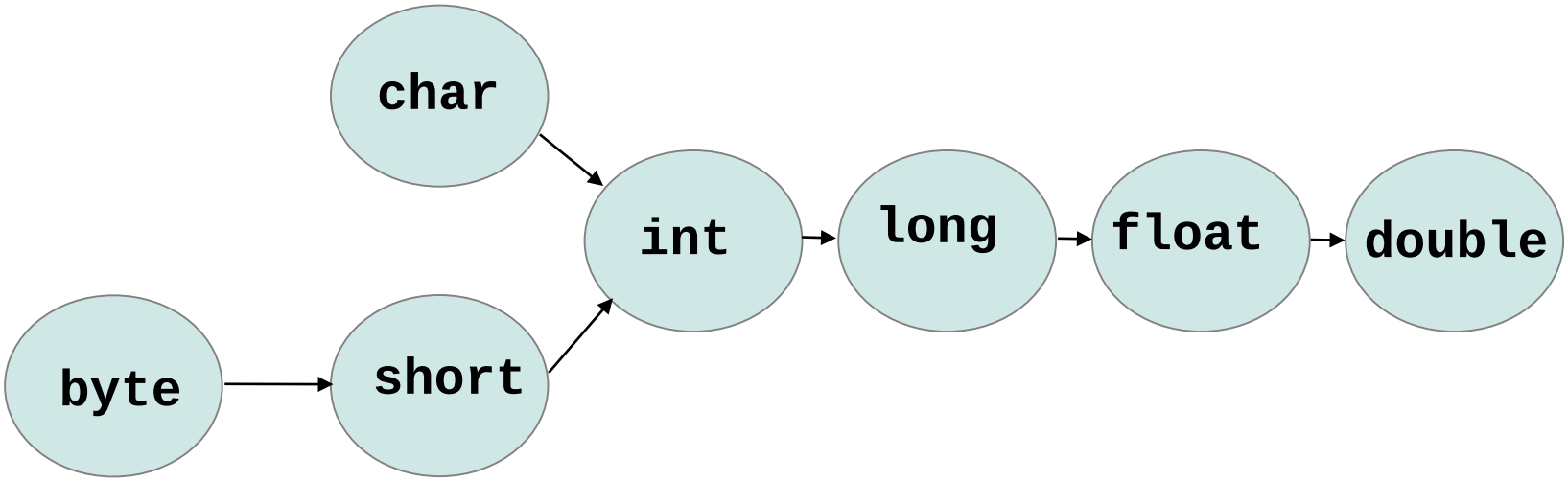
- The **`boolean`** data type has only two possible values : **`true`** or **`false`**
- The actual size can vary in different JVMs

```
boolean b1 = true;
boolean b2 = false;
```

Note! You cannot cast boolean to any type. The opposite is true as well.

Primitive data types casting

- The scheme of widening conversion:



- All other conversions are narrowing – you should use explicit casting:

```
double d = 5;
int i = (int)d;
```

Primitive data types

- By default, numeric literal is an **int** type value or **double** type value.

```
float a = 1.234; // Error
```

- However, the following is possible:

```
byte  b = 1;
short s = 2;
char  c = 3;
```

Note! Compiler checks the literal and if it is in the allowable range, allows for assignment

Exercise

Lab guide:

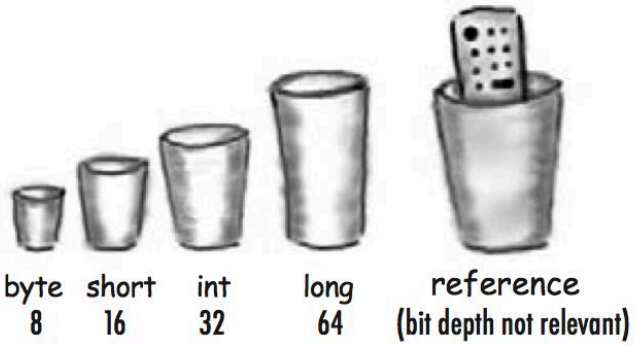
- Exercise 6

Object references

Object references

An object reference is just another variable value.

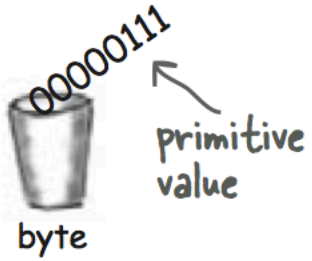
Something that goes in a cup.
Only this time, the value is a remote control.



Primitive Variable

```
byte x = 7;
```

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

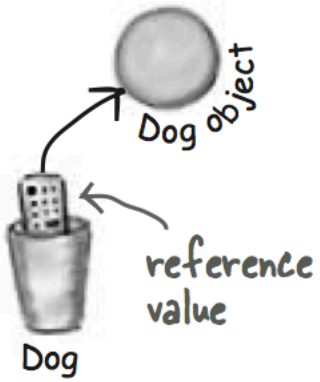


Reference Variable

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

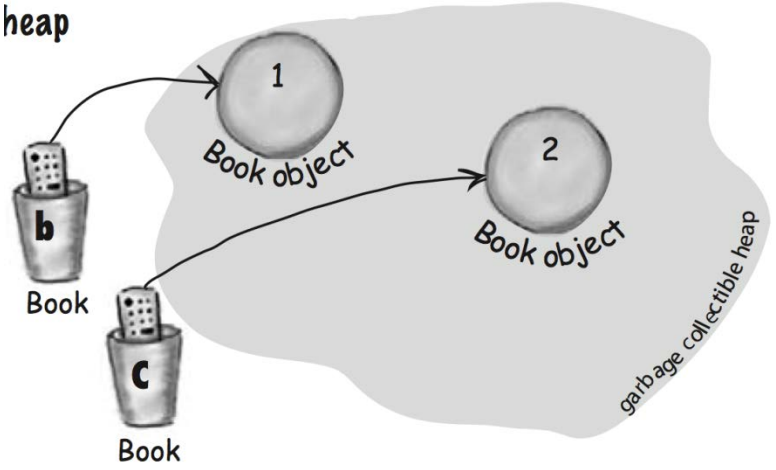
The bits representing a way to get to the Dog object go into the variable.

The Dog object itself does not go into the variable!

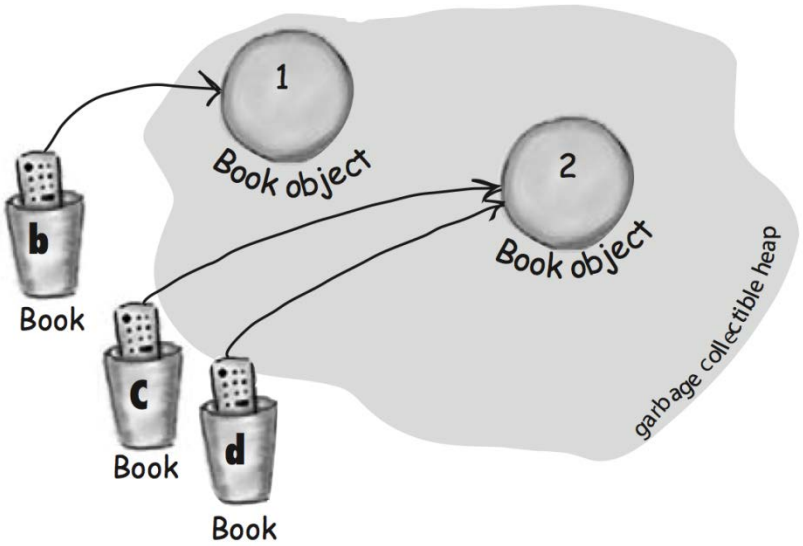


Life on the garbage-collectible heap

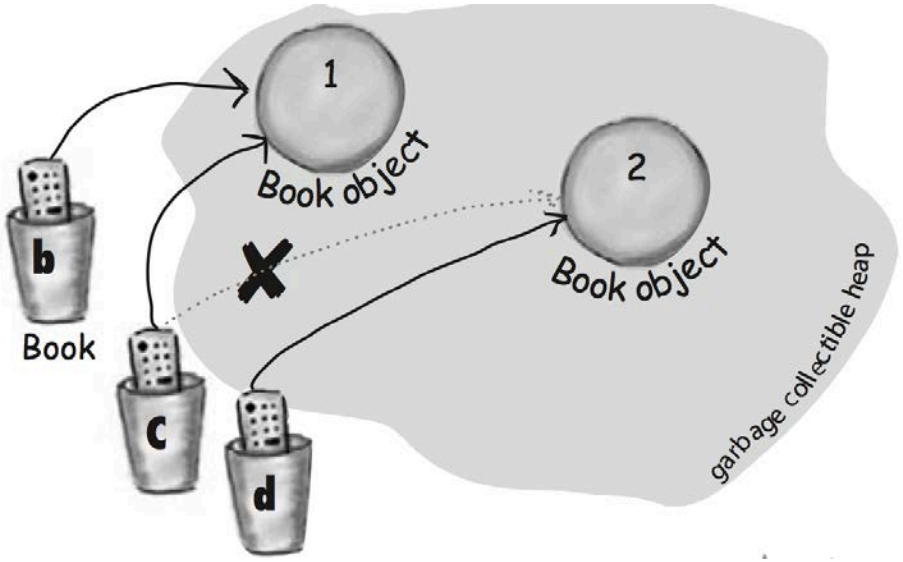
```
Book b = new Book();
Book c = new Book();
```




```
Book d = c;
```

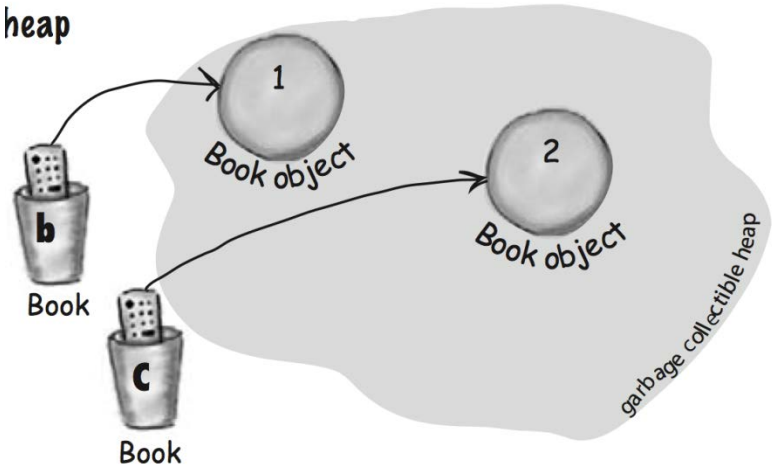



```
c = b;
```

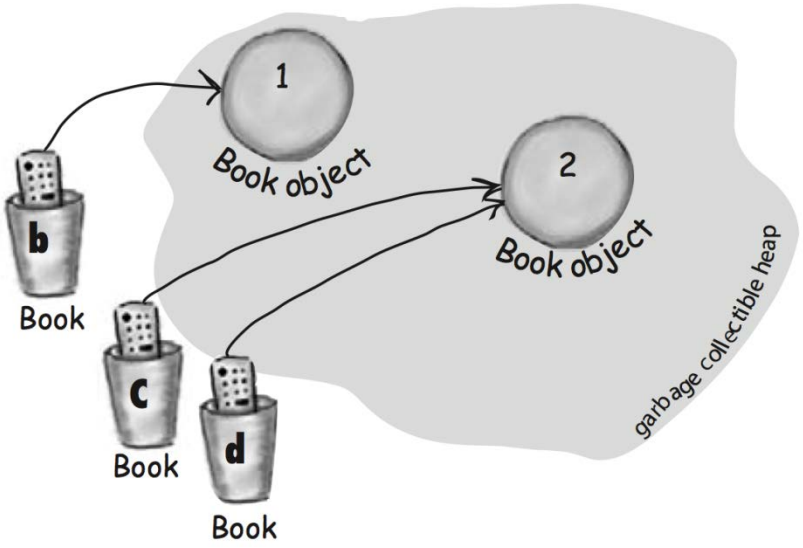


Life on the garbage-collectible heap

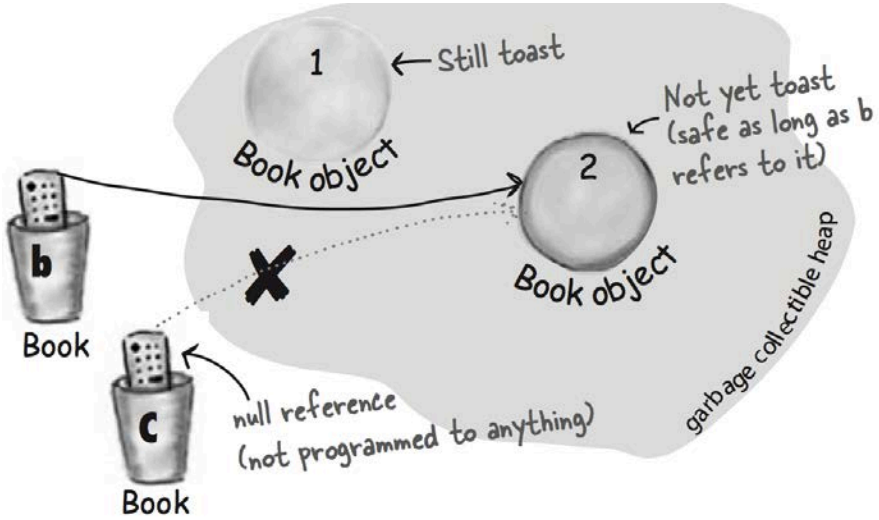
```
Book b = new Book();
Book c = new Book();
```




```
Book d = c;
```




```
b = c; c = null;
```



Arrays

Arrays of primitives

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

```
int[] nums;
```

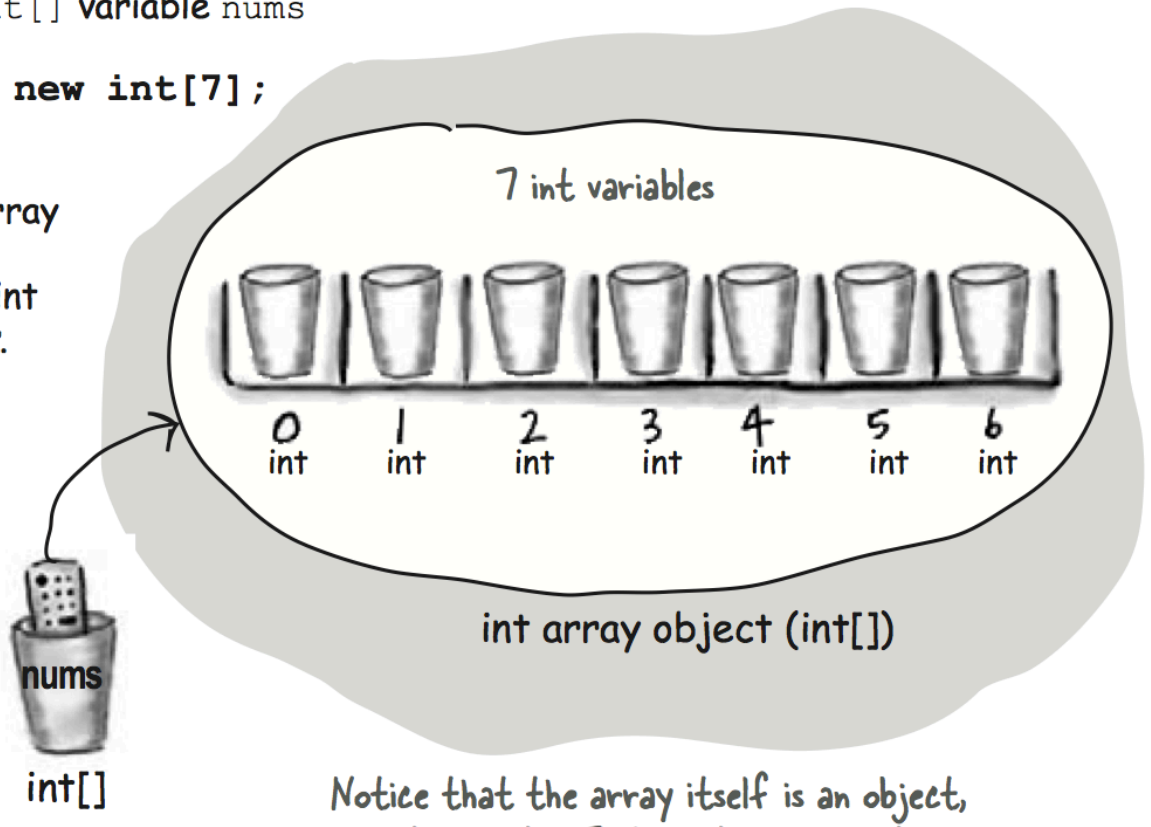
2. Create a new int array with a length of 7, and assign it to the previously-declared `int[]` variable `nums`

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

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

7 int variables

```
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.

Array of objects

Make an array of Dogs

1. Declare a Dog array variable

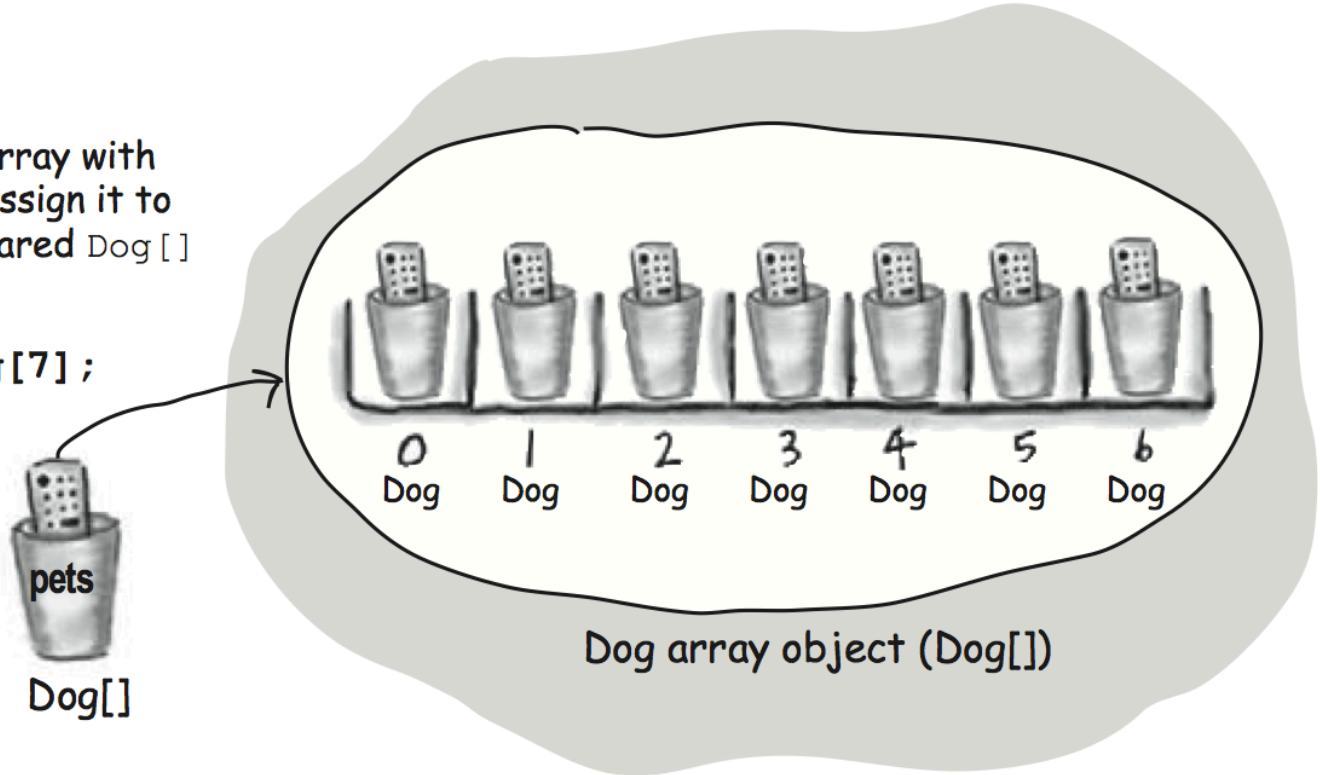
```
Dog[] pets;
```

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?

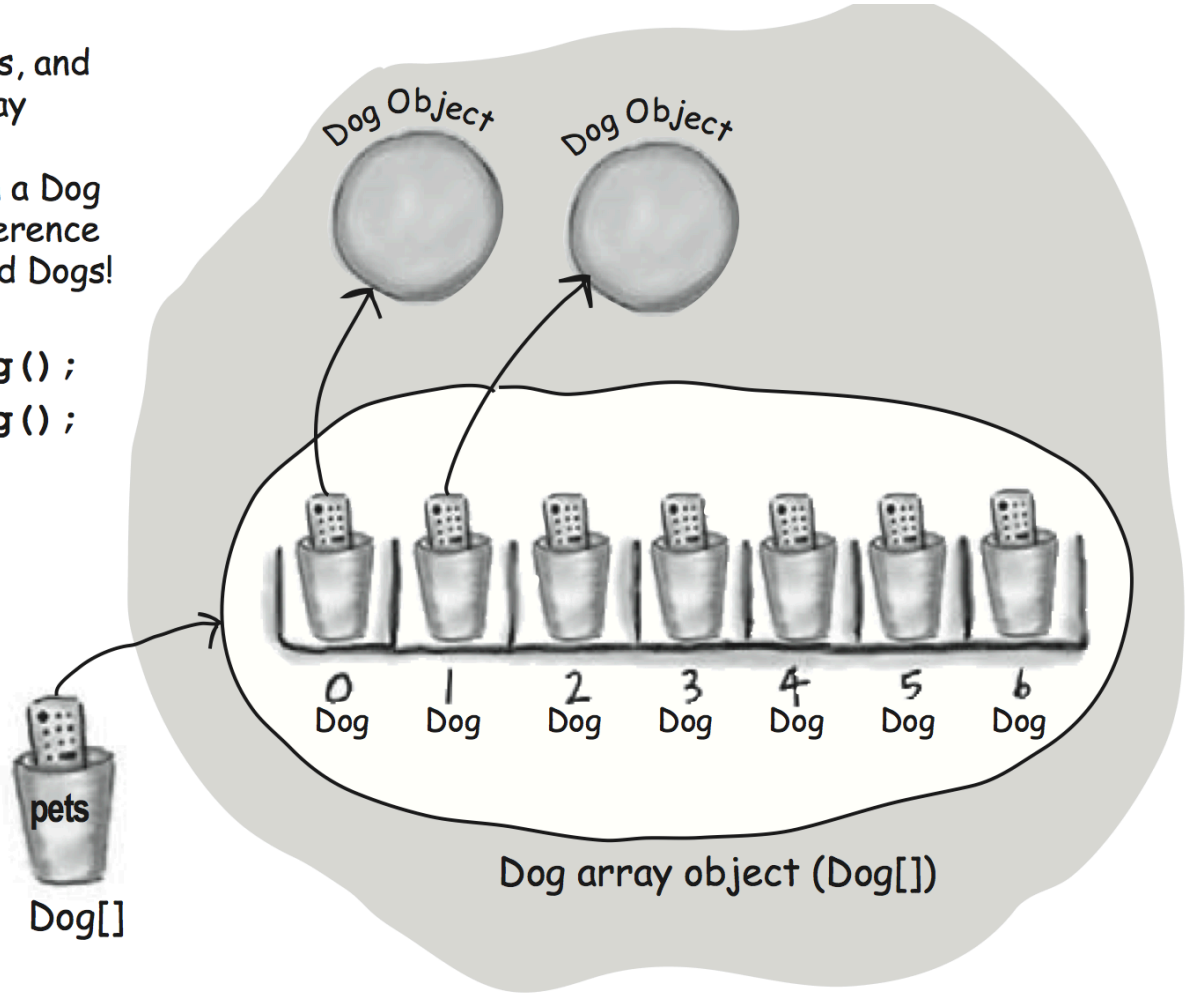
Dogs! We have an array of Dog *references*, but no actual Dog *objects*!



Array of 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();
```




```
Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Fido";
myDogs[0].bark();
```

Exercise

Lab guide:

- Exercise 7

Java operators

Java operators

Category	Operators	Associativity
Unary	++ -- + - ! ~ *(type)*	R to L
Arithmetic	* / % plus; -	L to R
Shift	<< >> >>>	L to R
Comparison	< <= > >= instanceof = = !=	L to R
Bitwise	& ^ |	L to R
Short-circuit	&& ||	L to R
Conditional	?:	R to L
Assignment	= *op*=	R to L

The + operator with strings

```
int i = 5;
int j = 5;

System.out.println(i + j); // 10

System.out.println("7" + j);// 75

System.out.println(new Person() + " 7"); // Person@ad3ba4 7

Person p = null;

System.out.println(p + "7"); // null7

System.out.println(new Person() + 7); // Compilation error
```

instanceof operator

- The **`instanceof`** operator tests object class in runtime.
- Left operand is a reference to an arbitrary object.
- Right operand is a class, interface or an array.

```
Object o = new String(“aaa“);
if (o instanceof String)
{
    System.out.println(“It's a String”);
}
```

instanceof operator

- **instanceof** operator can be used to test if an object is of a specified type:

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee

public void doSomething(Employee e) {
   if ( e instanceof Manager ) {
   // Process a Manager
   } else if ( e instanceof Engineer ) {
   // Process an Engineer
   } else {
   // Process any other type of Employee
   }
}
```

Break and continue with the label

```
for (int i = 0; i < array.length; i++) {
   if (array[i].secondString == null) {
      break;
   }
}
// break with label:
mainLoop: for (int i = 0; i < array.length; i++)
{
   for (int j = 0; j < array[i].length; j++) {
      if (array[i][j] == '\u0000') {
         break mainLoop;
      }
   }
}
```

Break and continue with the label

```
for (int i = 0; i < array.length; i++) {
    if (array[i].secondString == null) {
        continue;
    }
}
// continue to label:
mainLoop: for (int i = 0; i < array.length; i++)
{
    for (int j = 0; j < array[i].length; j++) {
        if (array[i][j] == '\u0000') {
            continue mainLoop;
        }
    }
}
```

Exercise

Lab guide:

- Exercise 8