CS0445: Algorithms and Data Structures 1

Lecture 1

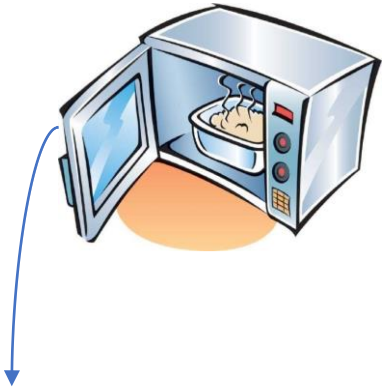
Java: reference variables

Marina Barsky: mbarsky@pitt.edu
people.cs.pitt.edu/~mbarsky/cs-0445

Java's approach to structuring data

- Java bundles data and actions within a single structure called ***object***
- Each object has properties and "knows" how to perform some actions
- Java program is a simulation of interacting objects that communicate by sending messages to each other

What sounds more natural?



```
cook (microwave, chicken)

microwave.cook (chicken)
```

New Data Types

- Java is a typed language, so each object must be of a specific ***type***
- We add new data types by defining a new ***class*** of objects
- The definition of a class contains:
  - instance variables (object properties)
  - method definitions (actions that the object can perform)

```
class  Microwave {
   int temperature;
   int power;

   void cook(Edible what){
   }
}
```

Encapsulation and data hiding

- We say that variables and methods are **encapsulated** within a class

- This encapsulation makes possible to set access restrictions: author of a class controls what data users can access directly

```
class Dog{
        String name;
        String breed;
        int size;
        double weight;

        void bark(){
            …
        }
}
```

Access modifiers

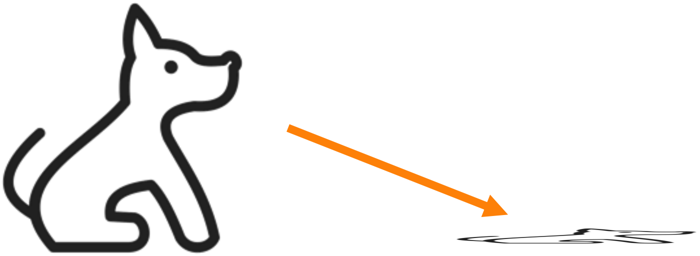
- These keywords control access of other classes to properties and methods within a given class
  - public – accessible outside class
  - private – inaccessible outside class
  - no modifier - accessible to all classes in the same package
  - protected – accessible only within class and subclasses [and same package]

```
class Dog{
    private String name;
    String breed;
    protected int size;
    double weight;

    public void bark(){
        if this.breed == …
    }
}
```

Only **public** methods/variables are accessible by the users of the class

Example: Bad Dog

```
public class BadDog {
    public String name;
    public int height;

    public void bark() {
        …
    }

    public static void main (String [] args) {
        BadDog d = new BadDog();
        d.height = 0;
    }
}
```



- We should never allow direct access to instance variables!

Example: Good Dog

declared
as private

```
public class GoodDog {
    private String name;
    private int height;


    public void setHeight (int h) {
        if (height > 0)
            height = h;
    }

    public int getHeight() {
        return height;
    }
}
```

setter

We reject (ignore)
invalid user input

getter

Data Hiding

- **Encapsulation** makes possible **data hiding**:
  - Declare instance variables as **`private`**
  - Use **`public`** methods to access/modify these variables

- The public methods for accessing object data are called:
  - *Accessors* (getters): get some value back
  - *Mutators* (setters): set value of some instance variable

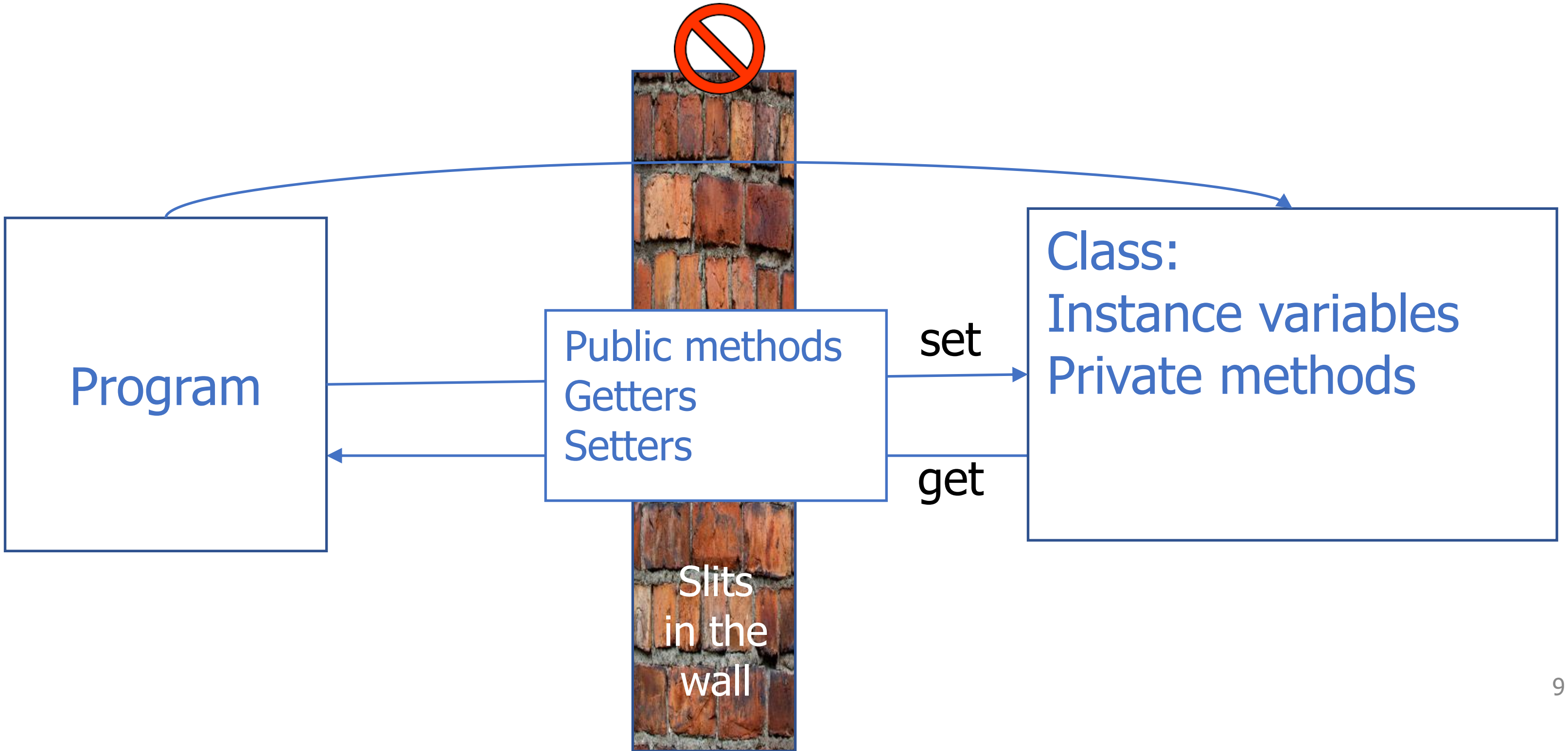
- With Data Hiding and Encapsulation we can:
  - validate the parameter passed to the method
  - reject unacceptable values (such as negative year): ignore them or throw an exception
  - replace the value with the closest valid or default value
  - change method implementation by changing the type of the inside storage and make it faster/safer without changing any code that uses our class

Protect object data: build the wall

- Programs that use your classes should NOT:

  be able to change the value of the instance variables directly

- Restrict the access to an object's data so you can only get it or change it by using methods



Program

Public methods
Getters
Setters

set

get

Class:
Instance variables
Private methods

Slits in the wall

Abstraction

- Abstraction - the process of extracting only essential property from a real-life entity
- In CS: Problem → storage + operations
- As a result of the process of abstraction we get a specification of data to be stored together with a set of operations on that data
- The user only needs to know the nature and the functionality of the data, and the public methods specs, but does NOT need to know/care about the implementation details:
  - What actual **data types are used** in the class
  - How the **methods are implemented**
  - These are declared private to the class – known to class author but not to class user

Encapsulation makes Abstraction possible

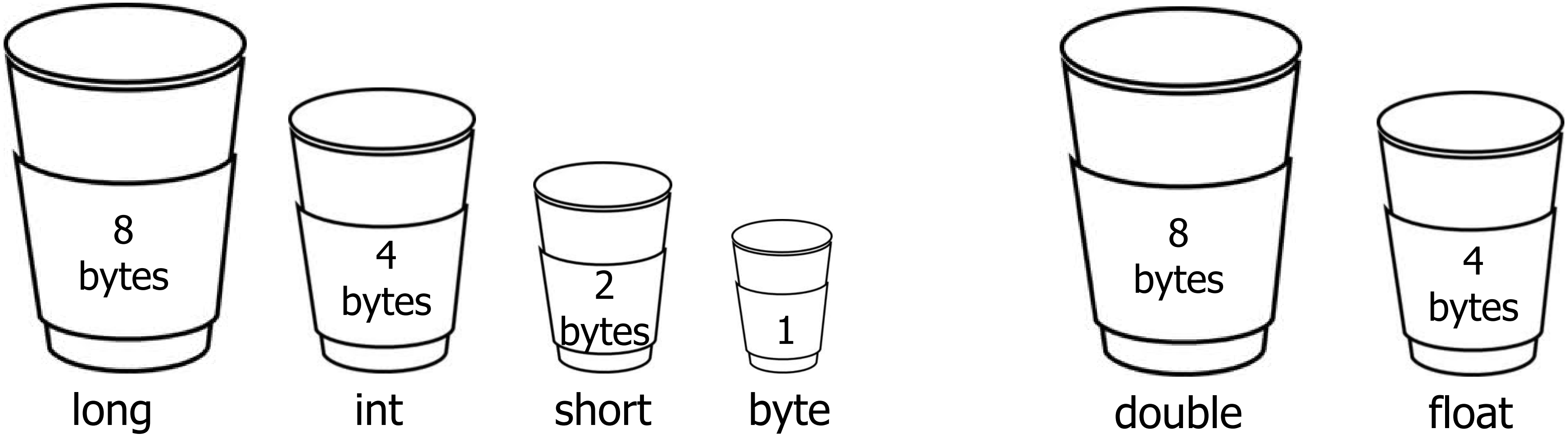
- Example: Java *BigInteger* class
- We hide the details and expose only properties and methods essential for using this class: we create an abstraction of Big Integer

- The encapsulation makes the abstraction possible by restricting access to the implementation details of a class

- Abstraction helps to manage the complexity of the software

Reference variables

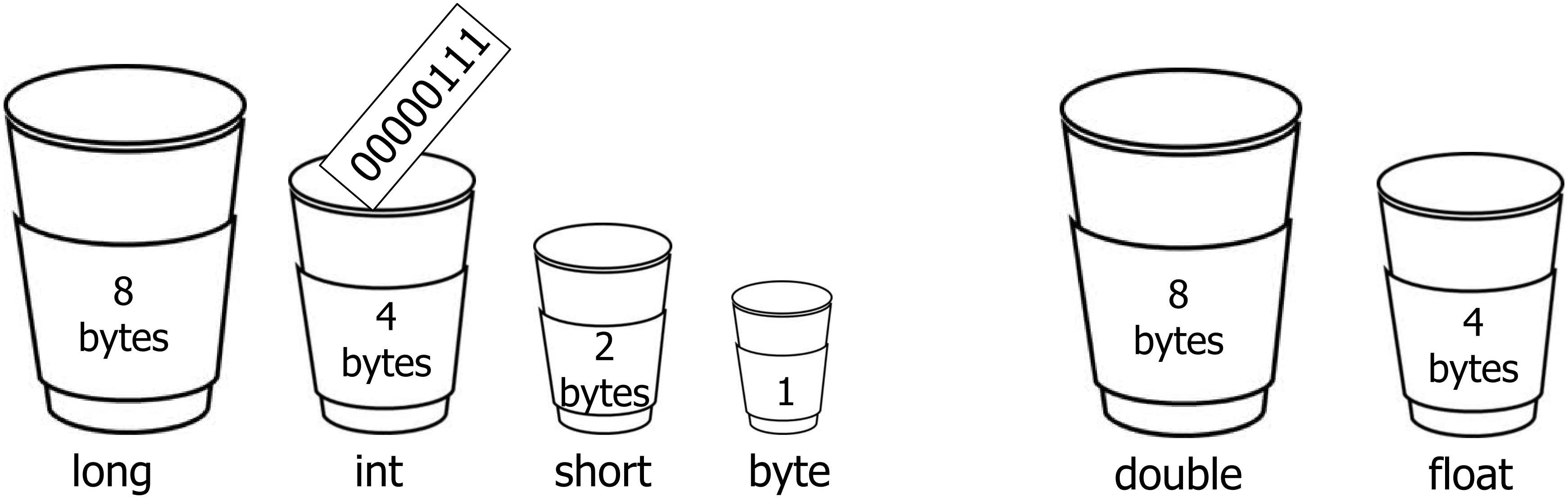
Recap: primitive numeric types



Integer

Floating point

Recap: primitive numeric types



Integer

Floating point

```
int i = 7;
```

Variables to hold objects

- With a new class of Objects – we create a new data type
- How do we declare a variable of a new type – what the size of a cup should be?

Reference variables

- A special reference variable does not hold the object itself, but it holds something like a pointer (or an address)
- Size of reference variables is the same for a given operating system: for example, it is long for 64-bit system

- In Java we don’t really know the value stored in the reference variable – no pointer manipulation is allowed
- And the Java Virtual Machine knows how to use the reference to get to the actual object

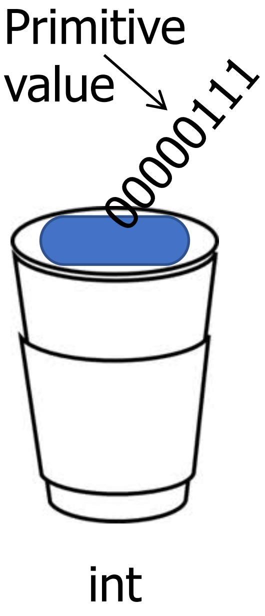
Reference and value

- An object reference is just another variable value: the value of an address
- Something that goes into the cup.

Primitive variable:

```
int x=7;
```

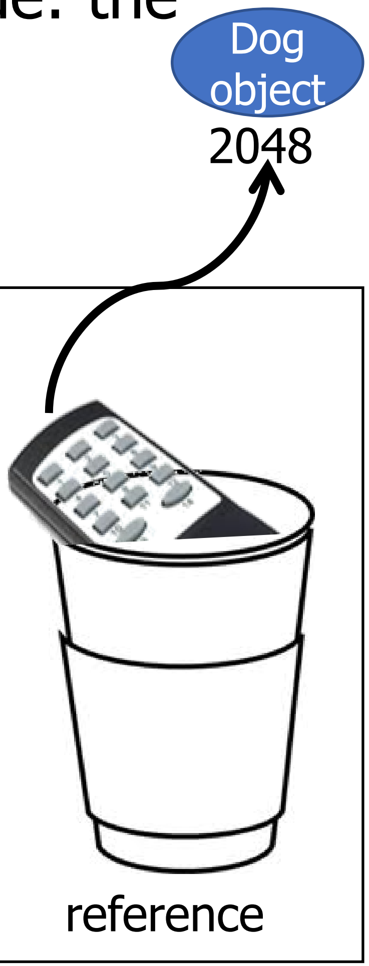
The bits representing 7 go into the cup



Reference variable:

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

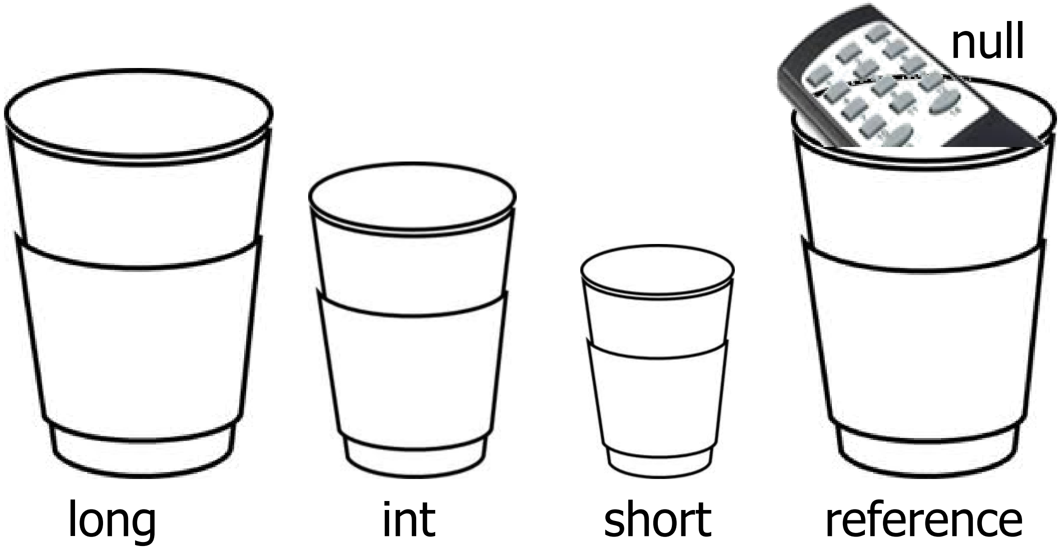
The bits representing the memory location of the Dog object go into the cup



Declaring Reference Variables

Dog myDog;

- reference variable of type *Dog*
- does not reference any actual object yet
- has default value *null*
- cannot call any methods of Dog class



Instantiating reference variable: creating an actual object

- All objects in Java are allocated dynamically
- Memory is allocated using the new operator
- Once allocated, objects exist for an indefinite period of time
  - As long as there is an active reference to the object

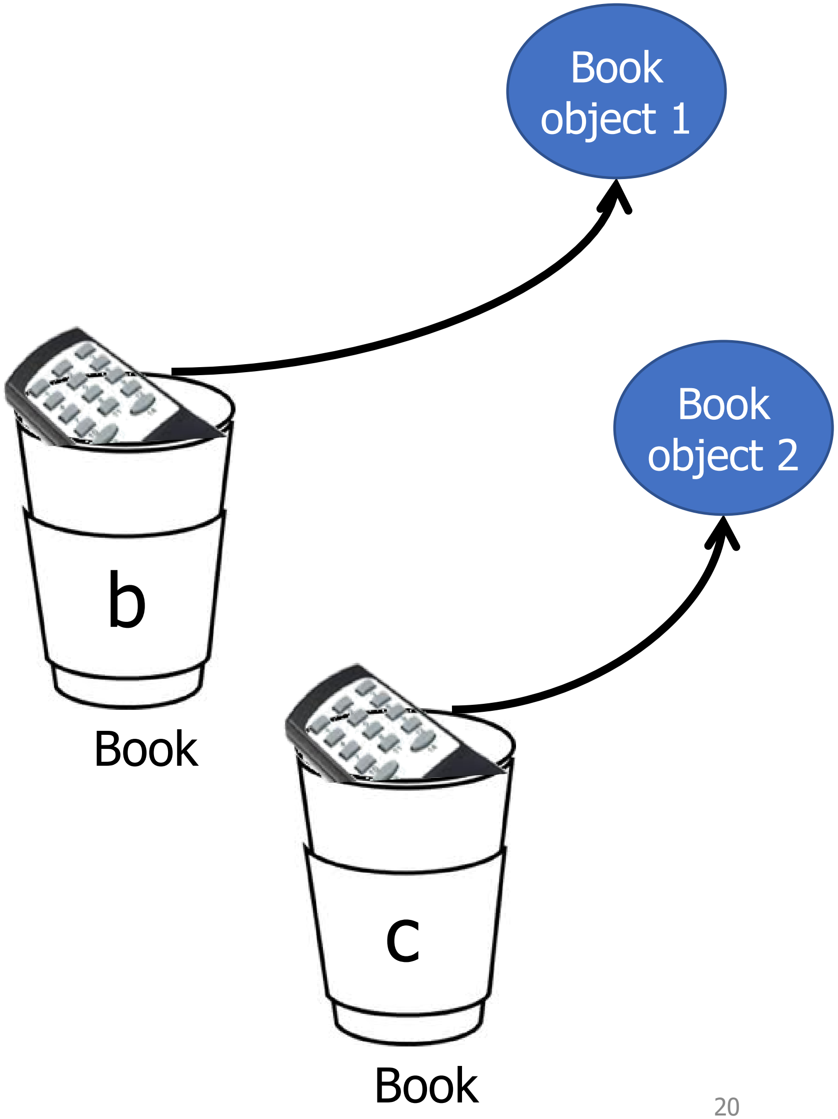
Now we can call the methods of class Dog

myDog=new Dog();

myDog.bark();

Example 1: assigning references

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

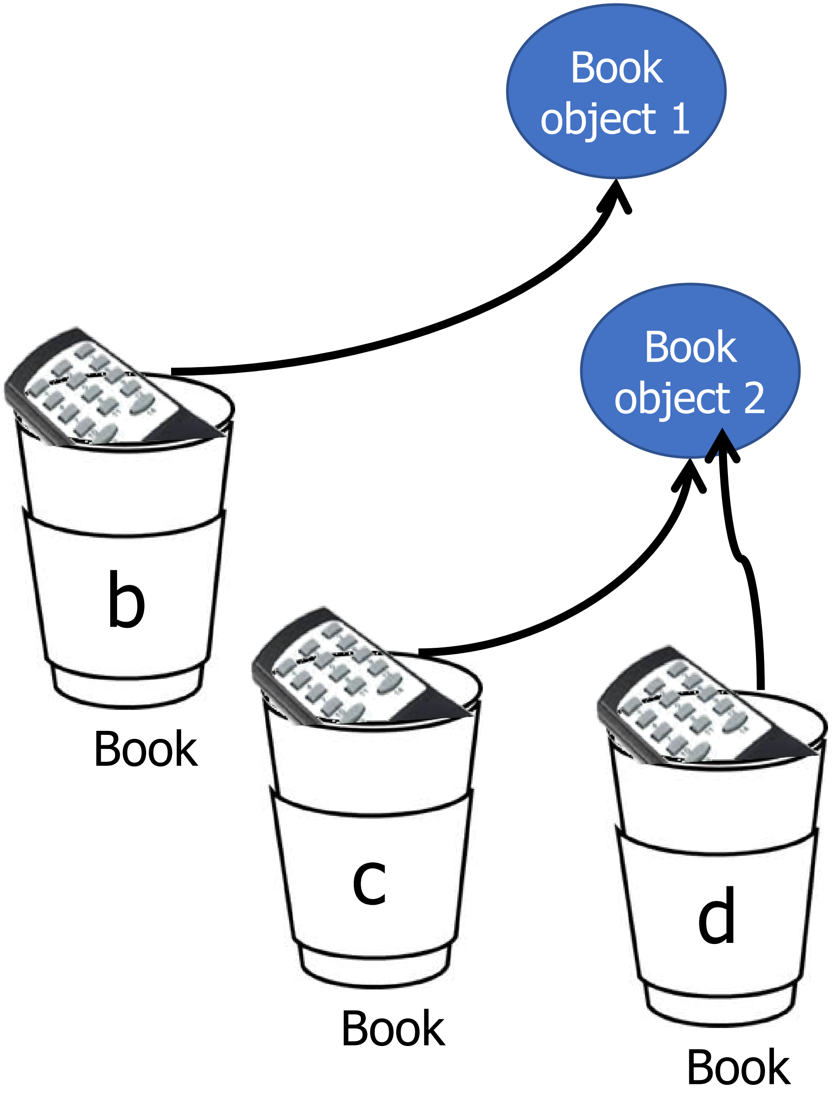


References: 2

Objects: 2

Example 1: assigning references

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



References: 3

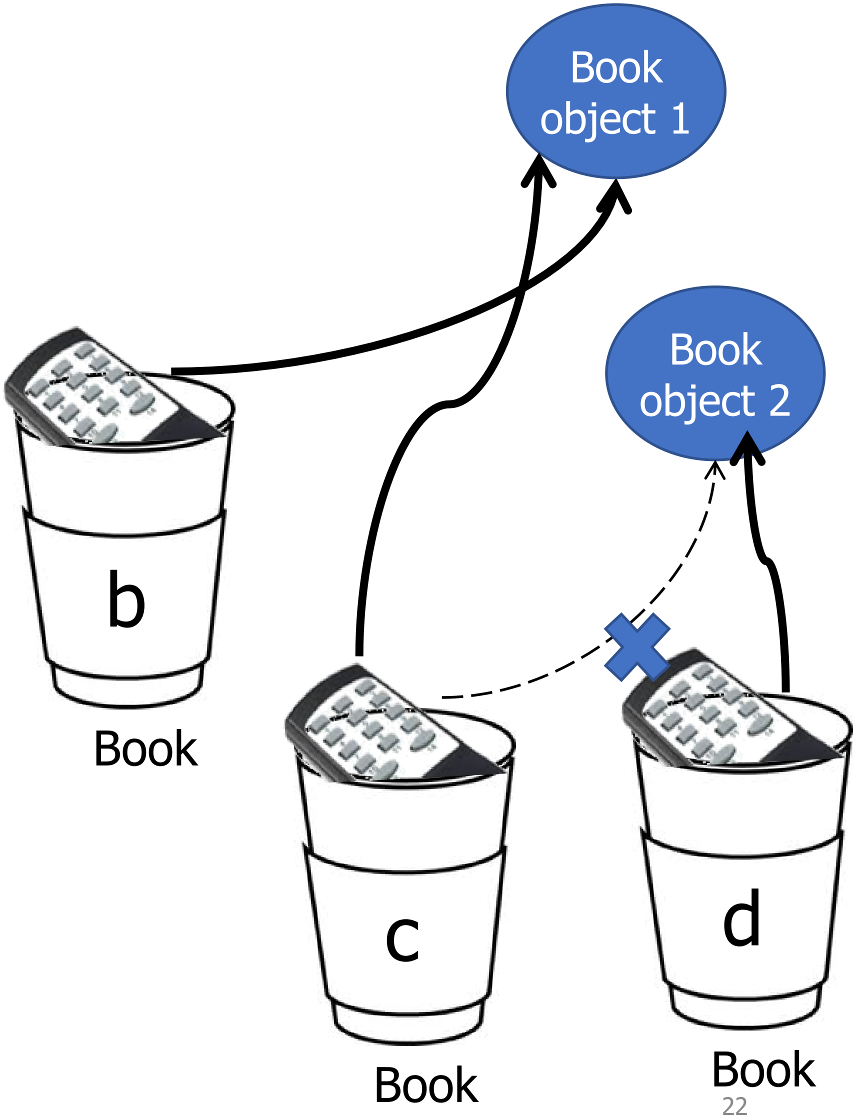
Objects: 2

Example 1: assigning references

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

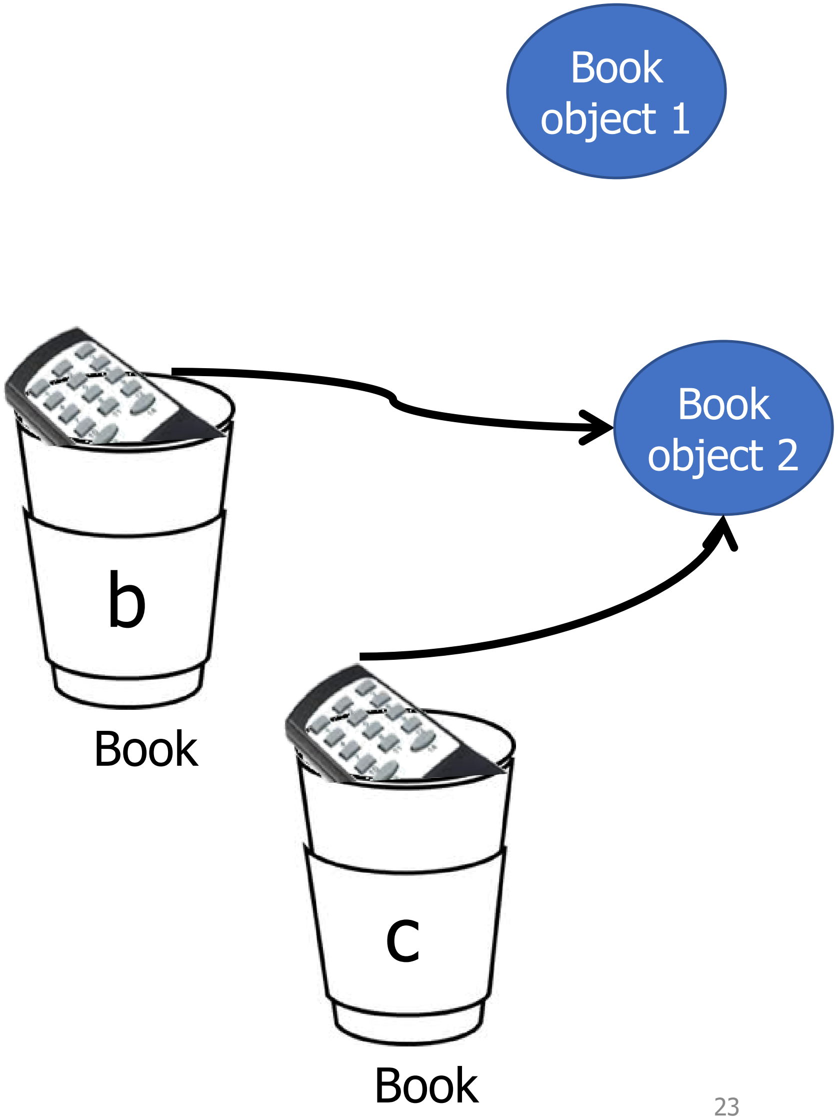
References: 3

Objects: 2



Example 2: assigning references

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



References: 2

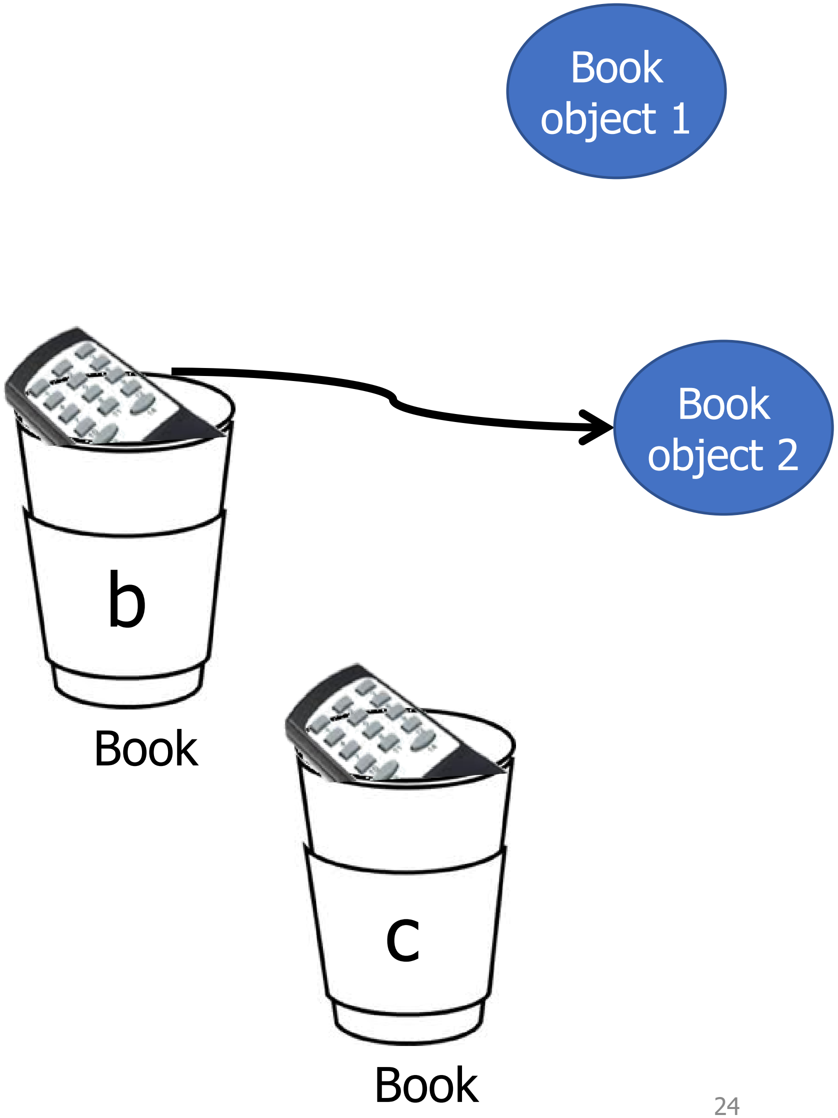
Reachable Objects: 1

Abandoned objects: 1

Example 2: assigning references

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

Active References: 1
Null references: 1
Reachable Objects: 1
Abandoned objects: 1



Recycling abandoned objects

- Java provides a feature called a **garbage collector** that automatically discovers when an object is no longer in use and destroys it
- This is a process that runs in the background during program execution

- When the amount of available memory runs low, the garbage collector reclaims objects that have been marked for collection

- The garbage collector provides a higher level of insurance against the insidious problem of *memory leaks*

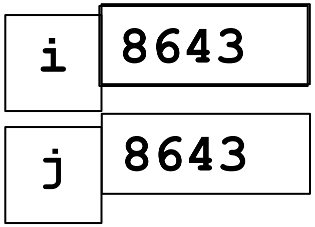
Notes to class

- Most frequent word
- Piazza

Example: comparing reference variables

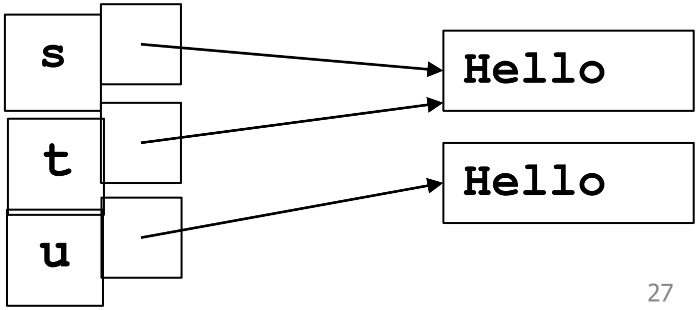
PRIMITIVE TYPE

```
int i;

i = 8643;

int j = i;


if (i == j)  true
```



REFERENCE TYPE

```
String s;

s = new String("Hello");

          [s = "Hello";]

String t = s;
String u = new String(s);

if (t == s) ...  true
if (u == s) ...  false
```



Reference comparison

- Know what you want to compare: references or contents

- For reference variables, we use a method to compare contents
  - ex. for strings, equals()
  - u.equals(s) returns true

- We can redefine equals() for our own classes

```
String s;

s = new String("Hello");


String t = s;
String u = new String(s);
```

Reference variables: summary 1/2

- Assignment creates aliases – references that refer to the same object

```
p1 = p2;
```

- Comparison operator checks if two references *refer to the same object*

```
 (p1 == p2)
```

- ✖ Unlike in C/C++ we cannot perform mathematical operations (no pointer arithmetic)

```
p1 + p2    p1++
```

- We do not need dereferencing: just access internal fields or call methods of an object using the dot operator on a reference variable itself

```
String s = "Hello World!";
System.out.println(s.length);
```

Reference variables: summary 2/2

- ✖ If two objects are exactly the same but are located in different memory locations, comparing their references will yield *false*

```
(p1 == p2)
```

- You need to implement a special method *.equals()* to compare objects themselves rather than their location addresses

```
(p1.equals(p2))
```

- ✖ Assigning references only copies a memory location and **does not copy** the object

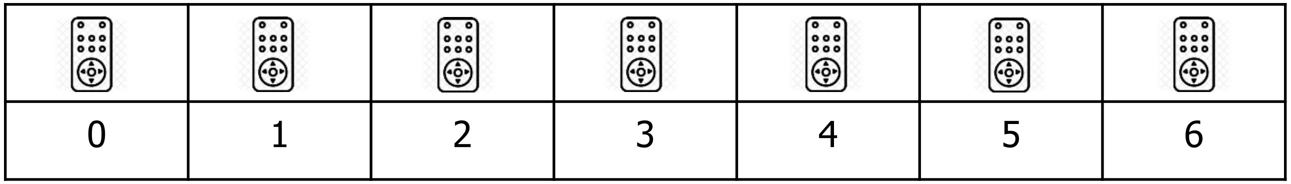
```
p1 = p2;
```

- You would need to implement the *.clone()* method to copy content of an object

```
p1 = p2.clone();
```

Array of References

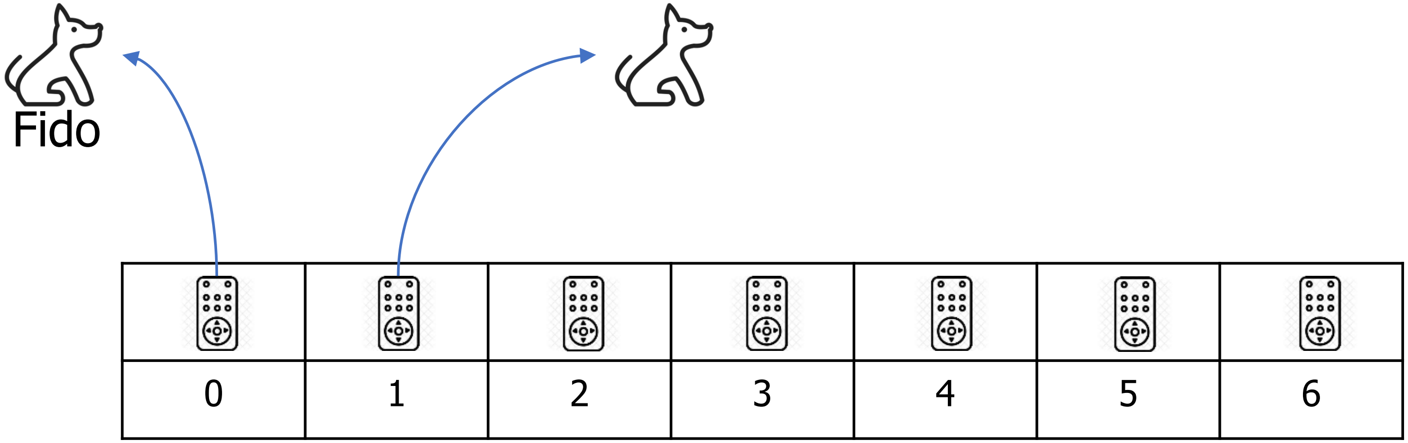
```
Dog pets = new Dog[7];
```



- This is array of references not array of dogs
- What is missing?
- Actual dogs!

How to create an array of Dogs

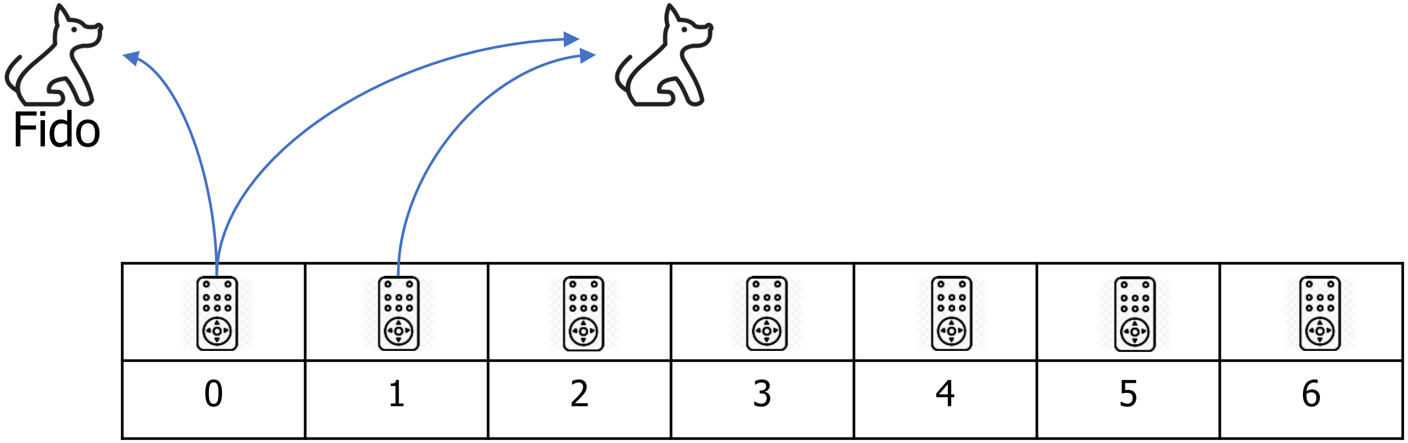
```
Dog pets = new Dog[7];
```



```
pets[0] = new Dog();
pets[1] = new Dog();
pets[0].name = "Fido";
```

How to create an array of Dogs

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



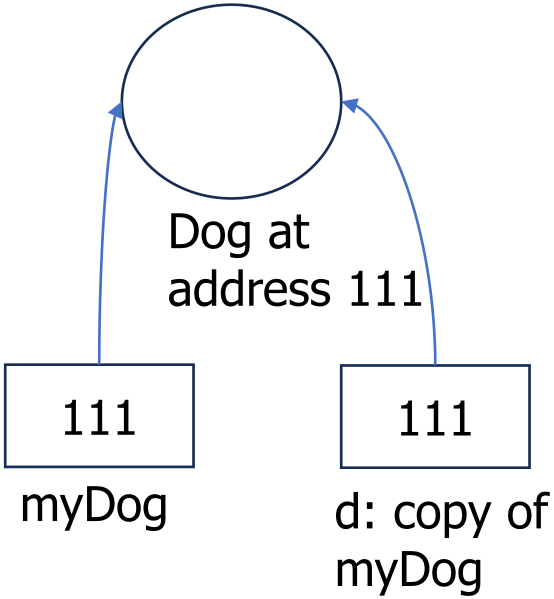
```
pets[0] = new Dog();
pets[1] = new Dog();
pets[0].name = "Fido";
pets[0] = pets[1];
```

- Who references “Fido”?
- What is stored in pets[2]?
- What is it pointing to?

Reference variables as method parameters

- Parameters in Java are **ALWAYS** **passed by value (by copy)**: only this time we copy the memory location!
- Thus inside the method we can manipulate the same object through a copy of the reference

```
public class Dog {
  int size;
}
```



```
public class Dogs {
  static void grow(Dog d){
       d.size ++;
  }
```

Manipulating the same object through a different reference

```
  public static void main (String[] args){
       Dog myDog = new Dog();
       myDog.size = 5;
       grow(myDog);
```

Copied myDog reference into a variable d

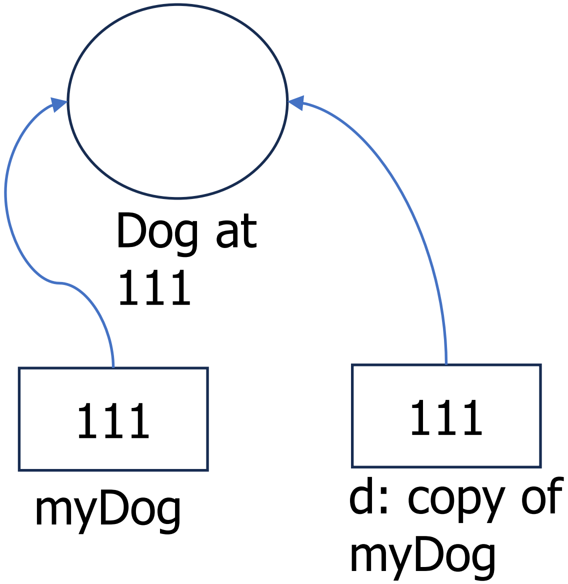
```
       System.out.println(myDog.size);
  }
}
```

myDog has size 6

Reference variables changing inside methods have no effect

- As long as reference parameter does not change an address stored in it, both the original and the copy refer to the same object and all the manipulations inside the methods are reflected in the original object

```
public class Dog {
   int size;
}
```

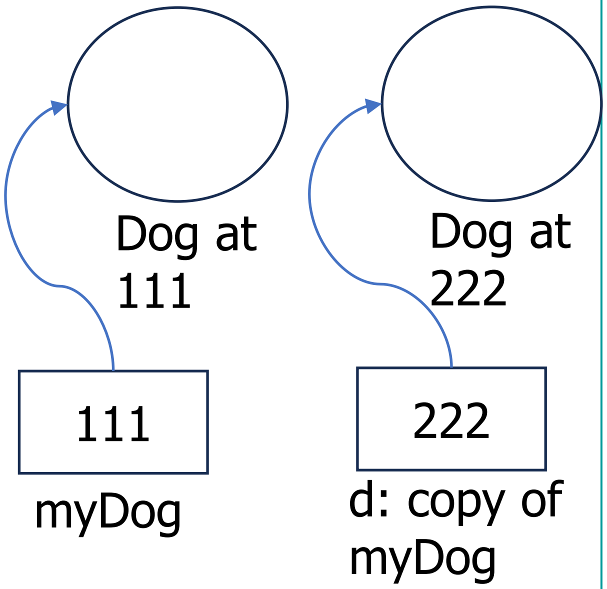


```
public class Dogs {
   static void reset(Dog d){
         d = new Dog();
         d,size = 0;
   }

   public static void main (String[] args){
         Dog myDog = new Dog();
         myDog.size = 5;
         reset(myDog);

         System.out.println(myDog.size);
   }
}
```

Reference variables changing inside methods have no effect

- Once we change the address in a copy of reference variable, everything we do to an object at this new address is not reflected in an original object

```
public class Dog {
  int size;
}
```



```
public class Dogs {
  static void reset(Dog d){
        d = new Dog();
        d,size = 0;
  }

  public static void main (String[] args){
        Dog myDog = new Dog();
        myDog.size = 5;
        reset(myDog);

        System.out.println(myDog.size);
  }
}
```

Manipulating a different object now

myDog has the same size 5

Demo

Dog.java
Example1.java

```java
public class Dog {
    String name;
    int size;
    public void bark() {
        String sound = "Ruff!";
        System.out.println(name +
                          " says " + sound);
    }
    public static void main (String [] args) {
        Dog d1 = new Dog();
        d1.name = "Bart";
        Dog [] pets = new Dog[2];
        pets[0] = new Dog();
        pets[0].name = "Lisa";

        pets[1] = new Dog();
        pets[1].name = "Marge";

        pets[0] = pets[1];
        pets[1].name = "Homer";
        pets[1] = d1;

        for(Dog d : pets)
            d.bark();
    }
}
```

- Homework test:
- What is printed?

A
```
Lisa says Ruff!
Homer says Ruff!
```

B
```
Homer says Ruff!
Bart says Ruff!
```

C
```
Lisa says Ruff!
Marge says Ruff!
```

D
```
Bart says Ruff!
Bart says Ruff!
```

E NONE OF THE ABOVE

```
public class Dog {
    String name;
    int size;
    public void bark() {
        String sound = "Ruff!";
        System.out.println(name +
                                 " says " + sound);
    }
    public static void main (String [] args) {
        Dog d1 = new Dog();
        d1.name = "Bart";
        Dog [] pets = new Dog[2];
        pets[0] = new Dog();
        pets[0].name = "Lisa";

        pets[1] = new Dog();
        pets[1].name = "Marge";

        pets[0] = pets[1];
        pets[1].name = "Homer";
        pets[1] = d1;

        for(Dog d : pets)
            d.bark();
    }
}
```

- Homework test:
- What is printed?

A
```
Lisa says Ruff!
Homer says Ruff!
```

[✓ B]
```
Homer says Ruff!
Bart says Ruff!
```

C
```
Lisa says Ruff!
Marge says Ruff!
```

D
```
Bart says Ruff!
Bart says Ruff!
```

E
```
NONE OF THE ABOVE
```

```
public class Dog {
    String name;
    int size;
    public void bark() {
        String sound = "Ruff!";
        System.out.println(name +
                           " says " + sound);
    }
   public static void main (String [] args) {
       Dog d1 = new Dog();
       d1.name = "Bart";
       Dog [] pets = new Dog[2];
       pets[0] = new Dog();
       pets[0].name = "Lisa";

       pets[1] = new Dog();
       pets[1].name = "Marge";

       pets[0] = pets[1];
       pets[1].name = "Homer";
       pets[1] = d1;

       for(Dog d : pets)
           d.bark();
   }
}
```

- How many references?

  3

- How many total objects?

  3

- How many abandoned objects?

  1

- What is the name of an abandoned Dog?

  “Lisa”

Q&A

Q&A

Q. Some of the terminology is unclear - instances, references, addresses, pointers, objects

- A. References (variables of type reference) in Java store a memory address of an object. In C/C++ the address is stored in a variable of type pointer. Both pointer and reference store the number: the memory address where they can find an object. The difference between a reference and a pointer is that to get to the place in memory from pointer we need to perform a dereferencing operation, but in Java we just use the reference variable as if it was an object itself (with dot). Every object is an instance of a specific type (or class).

Q. How to know whether I am dealing with the reference variables or the content of the reference variables.

- A. the content of a reference variable is always an address. If this address is not 0 (null), then using the dot operator on reference variable will change properties and call methods of an object at this memory location. The reference variable remains unchanged.

Q. Effects of Modifying Object Attributes vs. Reassigning References

- A. We can assign a new value to a reference variable in 2 ways only: either with keyword new or by assigning an address stored in another reference variable. With the new keyword JVM finds a free space in memory, builds all object compartments at this space, and returns an address of a new object to be stored in a reference variable. Modifying object attributes does not modify the reference (the address of an object in memory).

Q. assigning one variable another variable's value, then changing the assigned variable's value after the assignment, and how that affects the initial variable's value.

- A. It does not!

```
b = 4
a = b
b = 10
```

b is now 10, but a is still 4.

```
a = new String(“a”);
b = a;
a= new String(“aaa”);
```

The same works if a and b are reference variables which store numeric addresses: Originally address of string object “a” was 111, after assignment both a and b store 111. When a changes its value to address 222, this does not impact address stored in b – still 111.

Q. how arrays of reference variables specifically interact with memory

- A. Because reference variables are all of the same size, the arrays of addresses are not different from other arrays of numbers: they are allocated in memory consecutively.

Q. how we will be implementing new data structures and arrays

- A. Array is too fundamental to the operating system and memory management, so we are not going to reimplement it ☺. We might extend the arrays with more functionalities, for example we can implement dynamic arrays which grow on demand. We will implement data structures more complex than an array, which might contain arrays inside them or use linked structures instead of arrays.

Q. I don't understand how the machine knows how to allocate more space from object to object?

- A. That is performed by the operating system. In general, there are two types of memory: Stack and Heap. Stack is the place where all the primitive values are allocated, and the objects are allocated on the heap. Operating system keeps track which addresses in memory are not in use. When your program reaches the command *new*, the OS returns an address value, and it places an object with all its properties and code of methods in this memory location, occupying as much space as needed.

Q. Garbage Collection reclaiming objects?

- A. reclaiming here is in sense of reusing the place for new objects.

Q. I am still confused as to why instance variables should be declared as private.

A. If they are declared public, then any program that uses your class can modify them directly, and you cannot prevent the invalid values in your variables.

Consider variable height. If it is declared public, then it can be set to 0. The program has no place where to check that. Later in your program you compute the BMI index, and need to divide by height. You will get a run-time exception, which is not an acceptable program behavior. The object should not have accepted the zero value in the first place! The only way to check this is to declare height as private and to force the user of your class to change the height through a mutator (setter) method, in which you can put a check for validity.

Q. what is abstraction? Any application?

A. Abstraction in CS is a process of hiding the complexity of an entity behind a set of specs, which allow the specific use of a complex code without learning all the details. The whole of CS is based on the idea of abstraction. We will talk more about that when we discuss Abstract Data Types.