COMP 250
INTRODUCTION TO COMPUTER SCIENCE

Lecture 7 – OOD3 Object class and Type Conversion

Giulia Alberini, Fall 2018

Landing your Dream Tech Internship

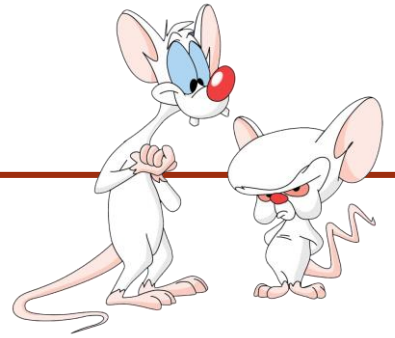
- Where: McMed 522
- When: Thursday, September 27th, from 6:30pm to 9pm
- What: Hear from past interns at Microsoft, Google, Amazon, and more about how to secure a software development internship! We will share tips and tricks about getting and nailing the interview.
- Facebook event: https://www.facebook.com/events/2405713899442986/?ti=ia

CS
US

FROM LAST CLASS

- Inheritance

WHAT ARE WE GOING TO DO TODAY?



- **The `Object` class**
- **Type Conversion**

Object CLASS

THE `Object` CLASS

- `Object` is the only class in java without a superclass. All other classes have one and only one direct superclass.
- In the absence of any other specific superclass, every class is implicitly a subclass of `Object`.

Class Object

java.lang.Object

`public class` **`Object`**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

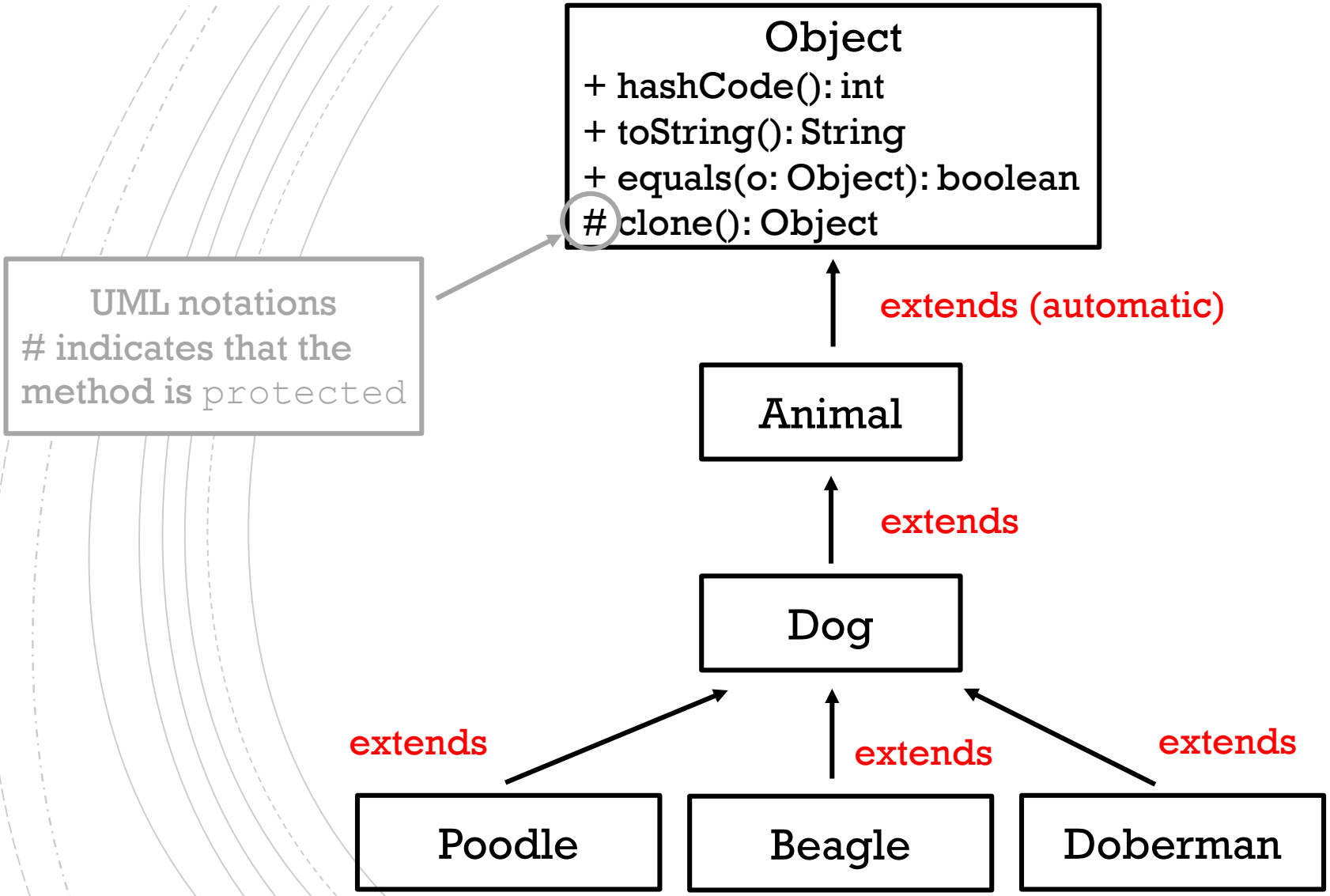
METHODS FROM `Object`

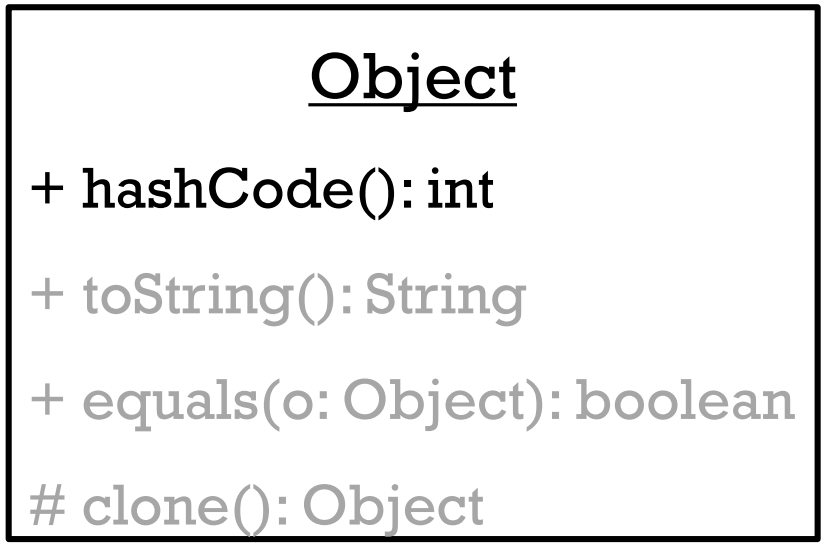
Here are some of the methods from the `Object` class:

`protected Object`	**`clone()`** Creates and returns a copy of this object.
`boolean`	**`equals(Object obj)`** Indicates whether some other object is "equal to" this one.
`protected void`	**`finalize()`** Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	**`getClass()`** Returns the runtime class of this `Object`.
`int`	**`hashCode()`** Returns a hash code value for the object.
`String`	**`toString()`** Returns a string representation of the object.

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

HIERARCHY FOR OUR EXAMPLES




Object
+ hashCode(): int
+ toString(): String
+ equals(o: Object): boolean
# clone(): Object

hashCode() - RETURN VALUE

- It returns a 32 bit integer associated to this object.
- "typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language".
- Use of hashCode() method : Returns a hash value that is used to search object in a collection.

`hashCode()` - REQUIREMENTS

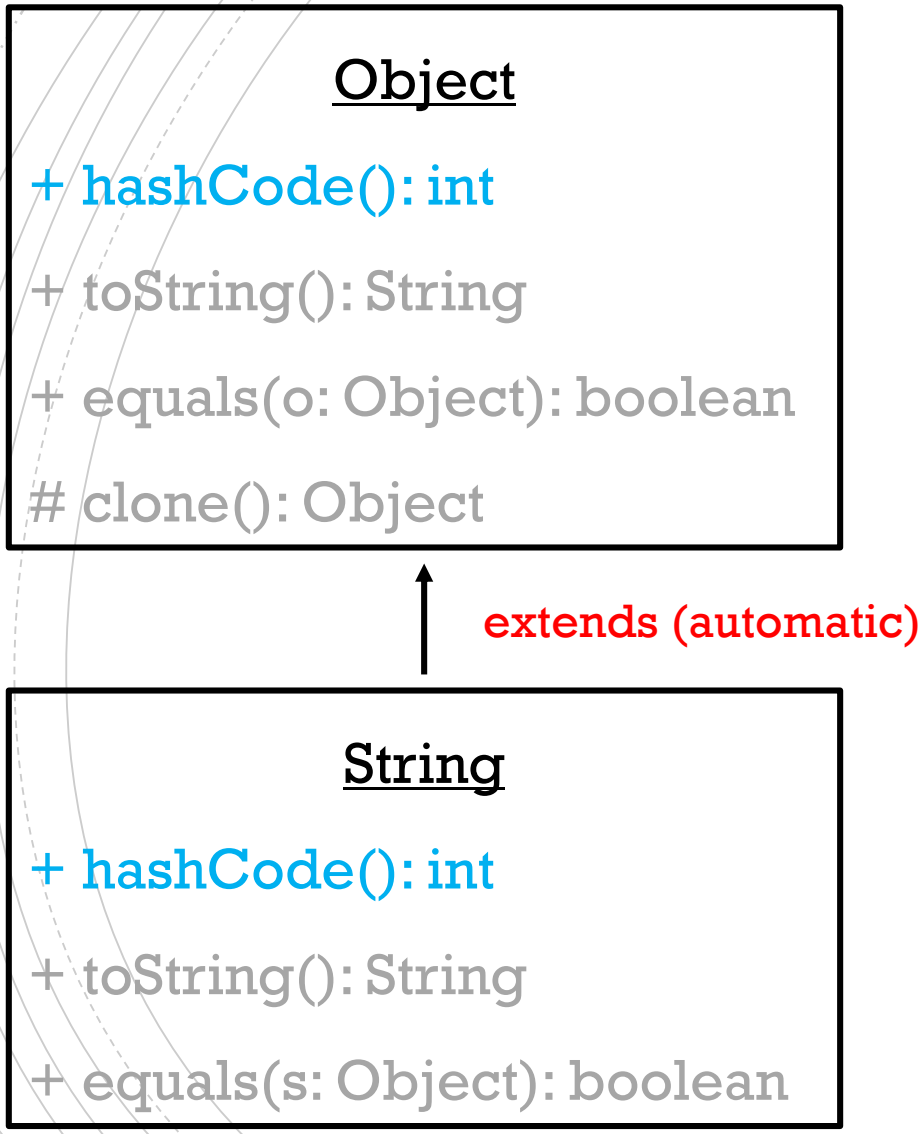
- "Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer."

- If `o1.equals(o2)` is `true`, then `o1.hashCode()==o2.hashCode()` should also be `true`.

  Note that the converse does not need to hold!

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--

EXAMPLE



The class `String`

overrides `hashCode()`

EXAMPLE

The method `hashCode()` **from the class** `String`

hashCode

```
public int hashCode()
```

Returns a hash code for this string. The hash code for a `String` object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using `int` arithmetic, where `s[i]` is the *i*th character of the string, `n` is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

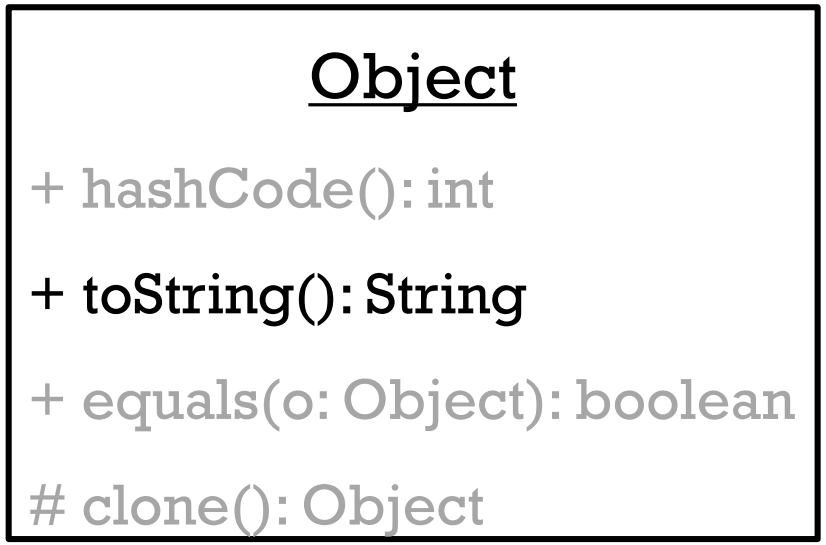
Overrides:

`hashCode` in class `Object`

Returns:

a hash code value for this object.

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html


Object
+ hashCode(): int
+ toString(): String
+ equals(o: Object): boolean
# clone(): Object

`toString()`

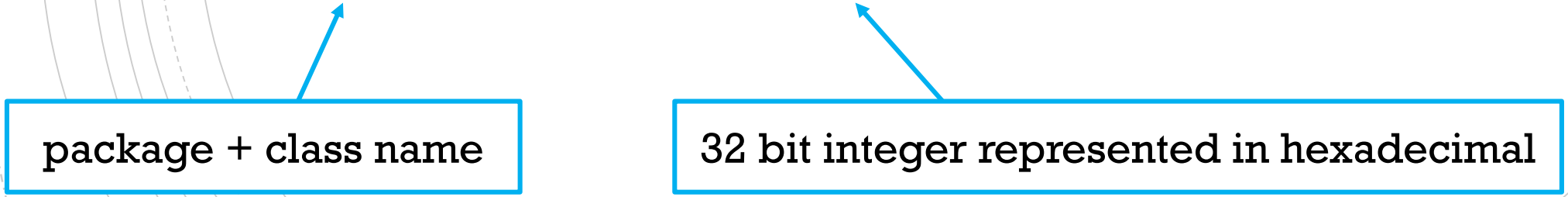
- Returns a string representation of the object.
- It is recommended that all subclasses override this method.
- The `toString()` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object.

EXAMPLE

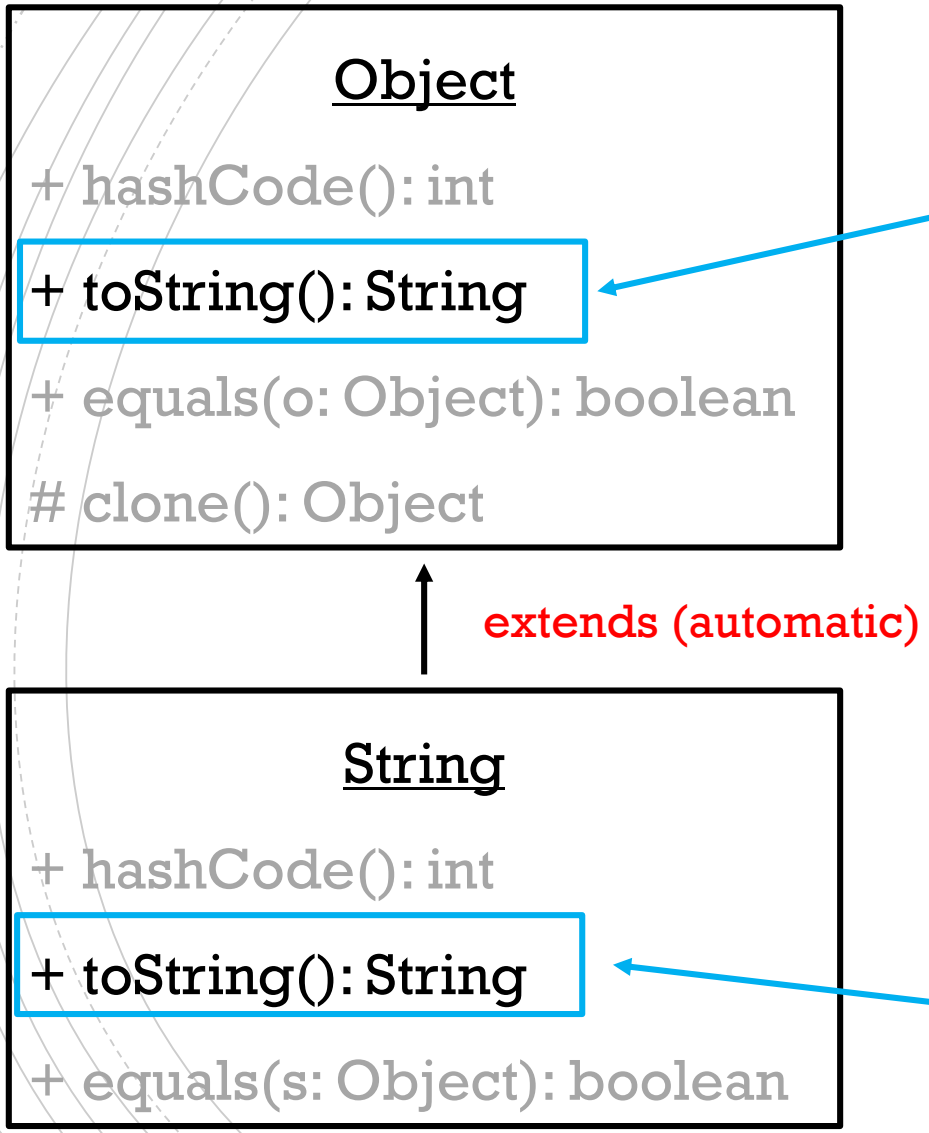
```
System.out.println( new Object() );
```

What does this print?

```
java.lang.Object@7852e922
```



EXAMPLE



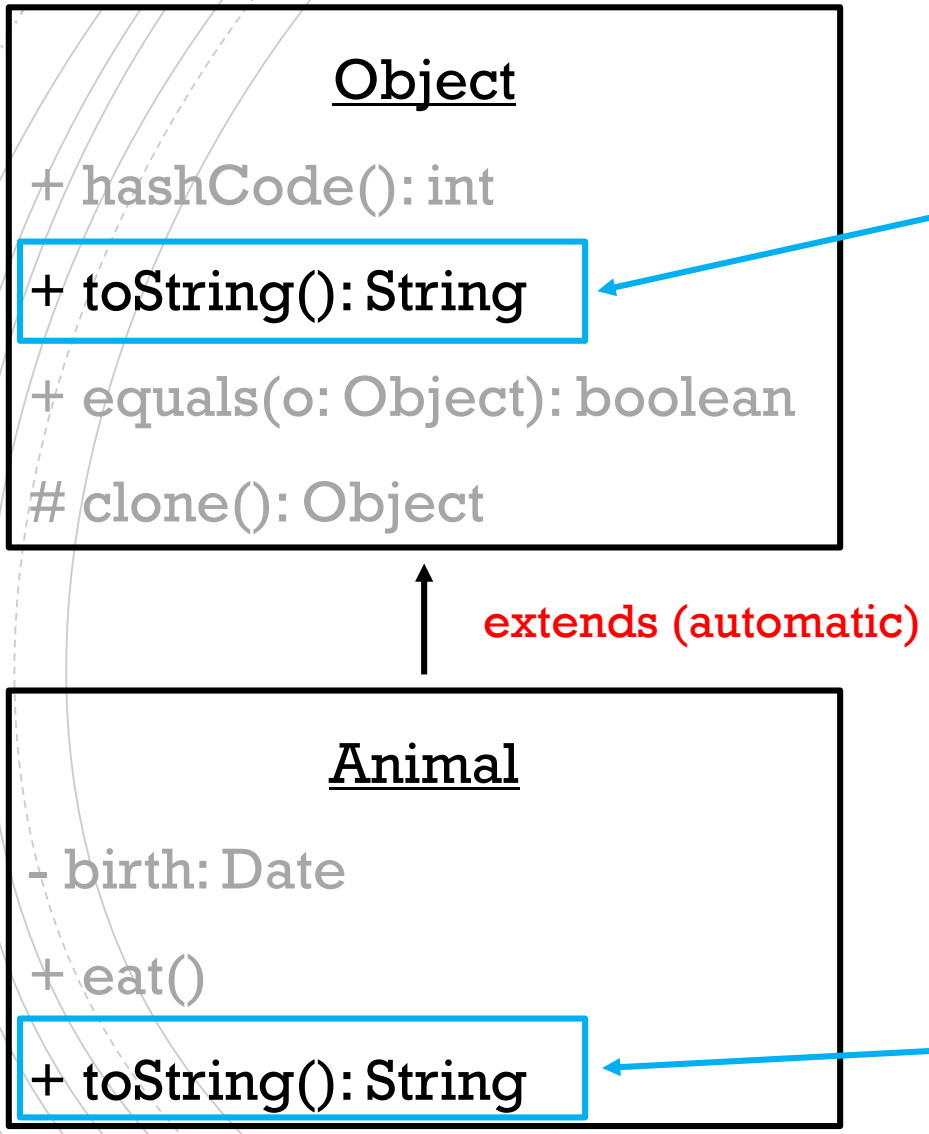
Returns the following:

```
className + "@" +
Integer.toHexString(hashCode())
```

`toString()` **is overridden in the class** `String`

Returns the object itself

EXAMPLE

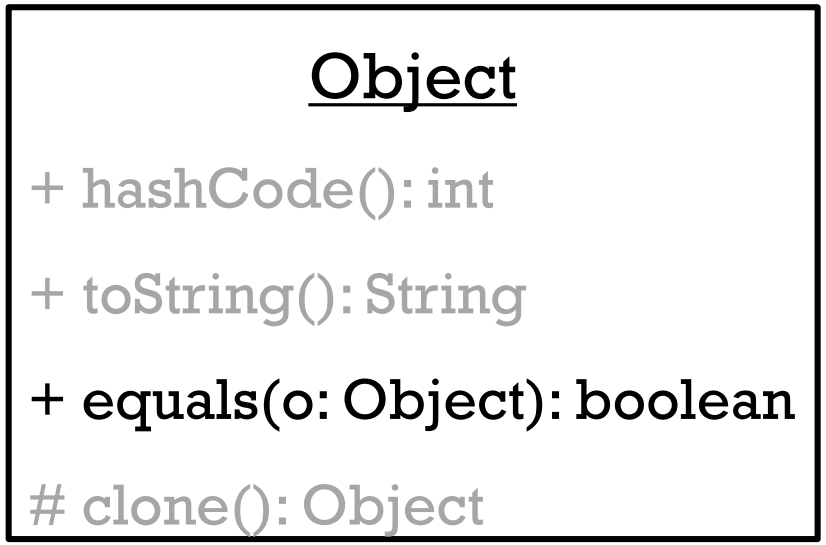


Returns the following:

```
className + "@" +
Integer.toHexString(hashCode())
```

`toString()` **is overridden in the class** `Animal`

Returns… depends on your implementation!


Object
+ hashCode(): int
+ toString(): String
+ equals(o: Object): boolean
# clone(): Object

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

equals()

equals

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

see MATH 240

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

`equals()` – IMPLEMENTATION

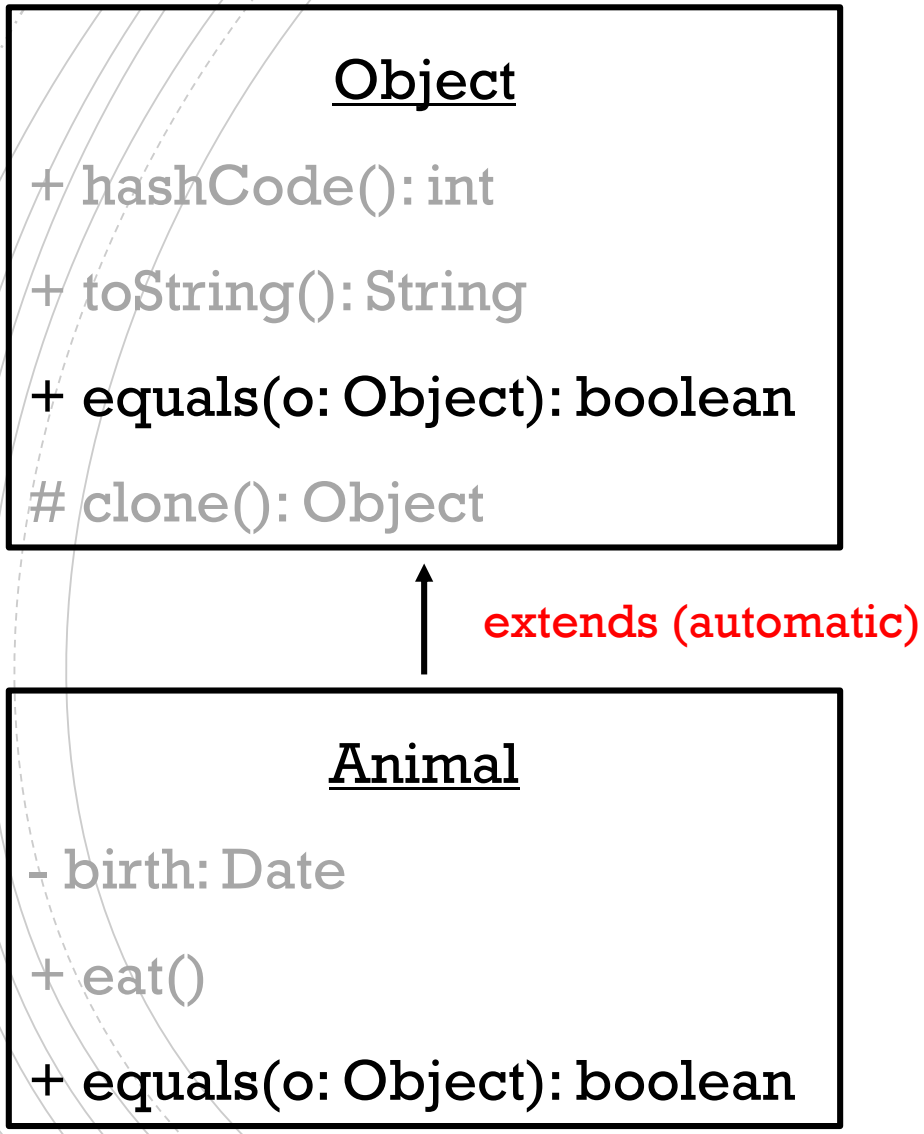
For any non-null reference values `obj1` and `obj2`,

`obj1.equals(obj2)` **returns** `true`

if and only if

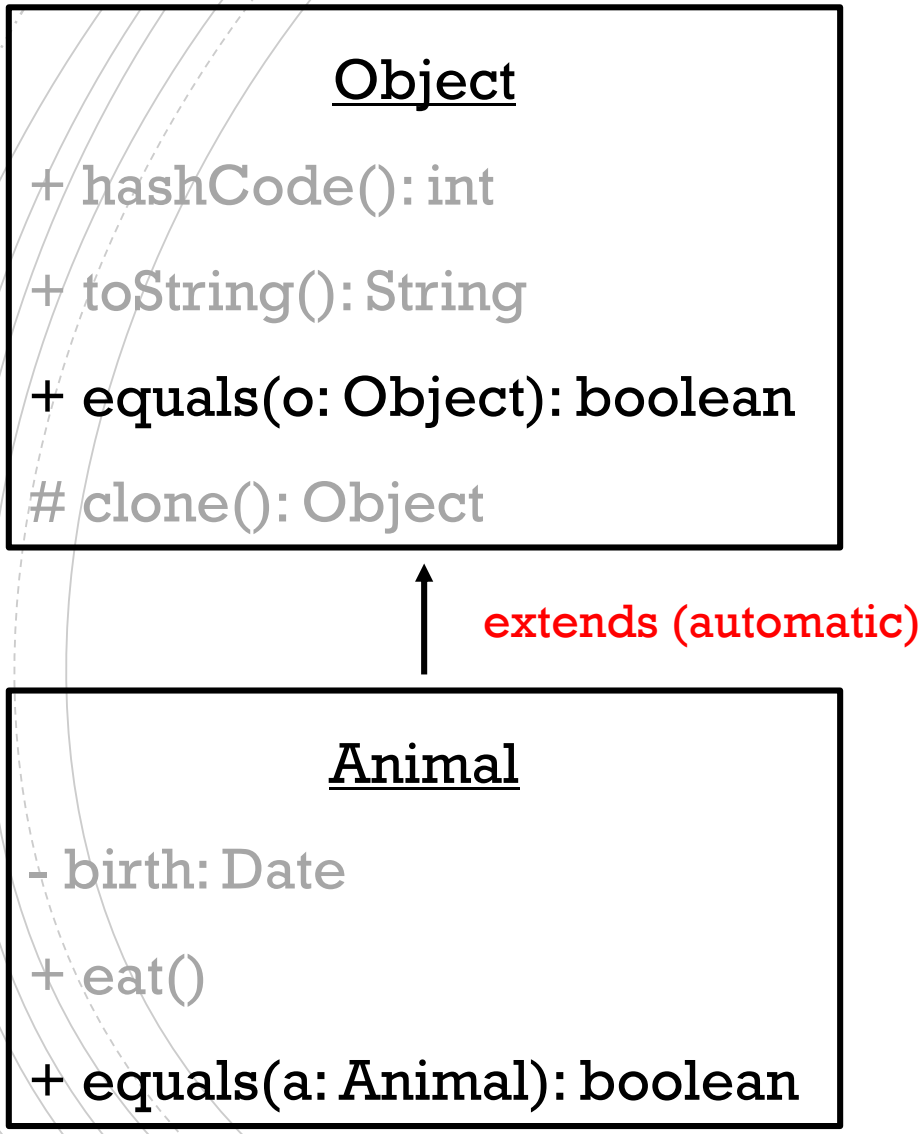
`obj1 == obj2` **has value** `true`

EXAMPLES



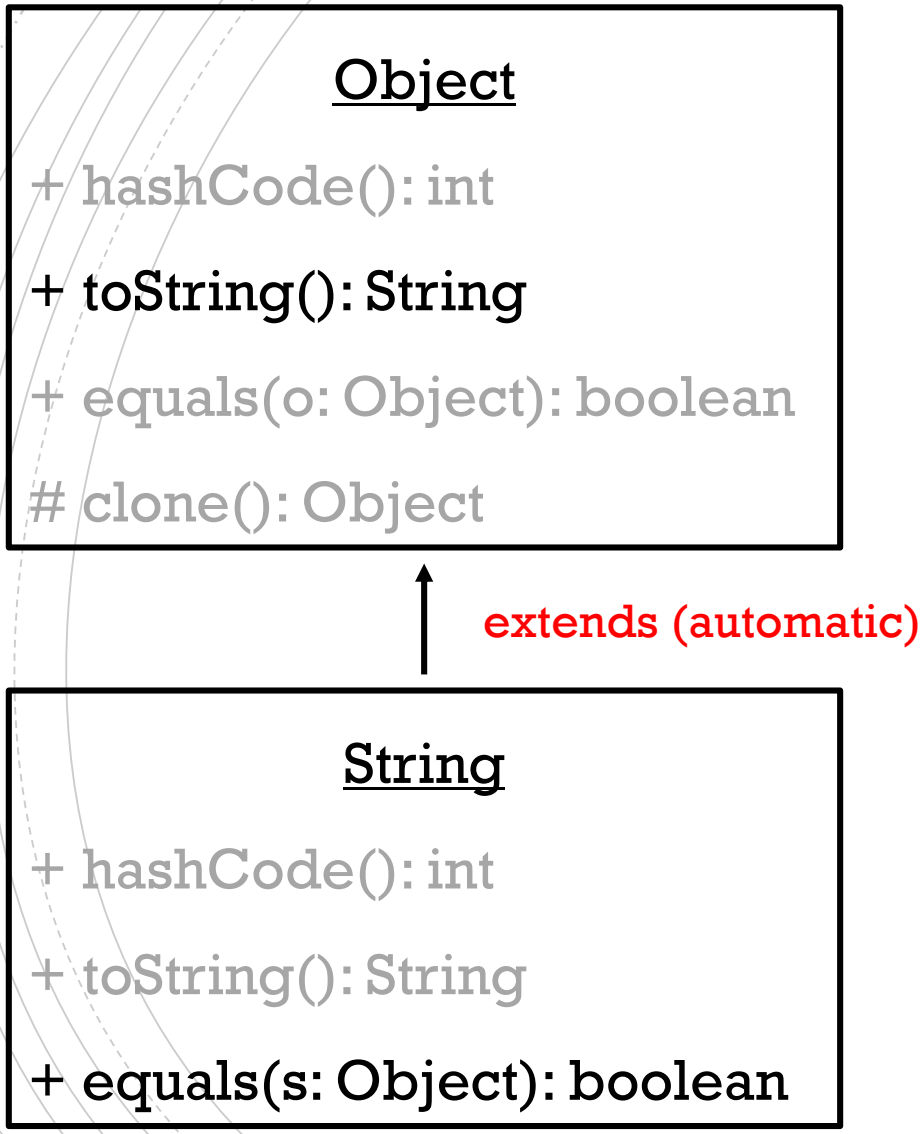
`Animal` **overrides the** `equals()` **method**

EXAMPLES



`Animal` overloads the `equals()` method

EXAMPLES



`String` **overrides the** `equals()` **method**

equals() FROM String

equals

```
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.

Overrides:

`equals` in class `Object`

Parameters:

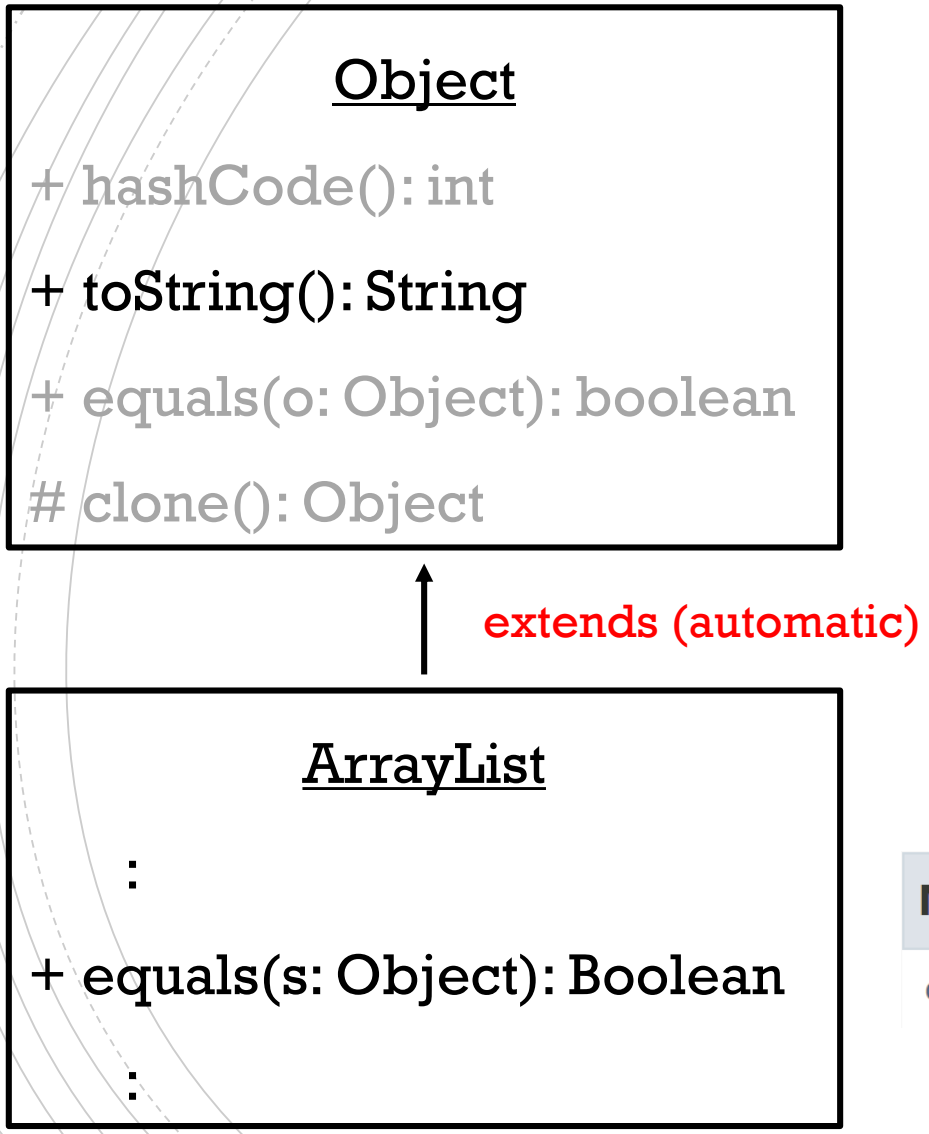
`anObject` - The object to compare this `String` against

Returns:

`true` if the given object represents a `String` equivalent to this string, `false` otherwise

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

EXAMPLES



`ArrayList` **inherits an overridden version of the** `equals()` **method**

Methods inherited from interface java.util.List

`containsAll, equals, hashCode`

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

equals() FROM List

equals

`boolean equals(Object o)`

Compares the specified object with this list for equality. Returns `true` if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are *equal*. (Two elements `e1` and `e2` are *equal* if `(e1==null ? e2==null : e1.equals(e2))`.) In other words, two lists are defined to be equal if they contain the same elements in the same order. This definition ensures that the equals method works properly across different implementations of the `List` interface.

Specified by:

`equals in interface Collection<E>`

Overrides:

`equals in class Object`

Parameters:

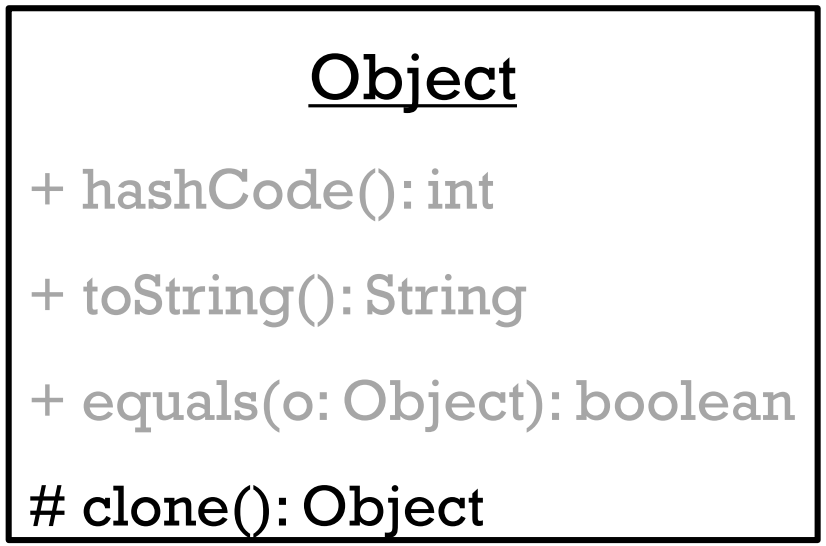
`o - the object to be compared for equality with this list`

Returns:

`true if the specified object is equal to this list`

TO LOOK FORWARD TO

- We will be talking more about interfaces like `List` in a couple of weeks!


Object
+ hashCode(): int
+ toString(): String
+ equals(o: Object): boolean
# clone(): Object

`clone()`

- **Creates and returns a copy of *this* Object**
- **Intent:**
  - `x.clone()` **and** `x` **points to objects of the same type.**
  - `x.clone() == x` **is** `false`
  - `x.clone().equals(x)` **is** `true`
- **By convention, when overriding** `clone`, `super.clone()` **should be called.**

EXAMPLE – HOW TO CLONE AN ARRAYLIST?

Suppose we have a `ArrayList` of `Shape`s.

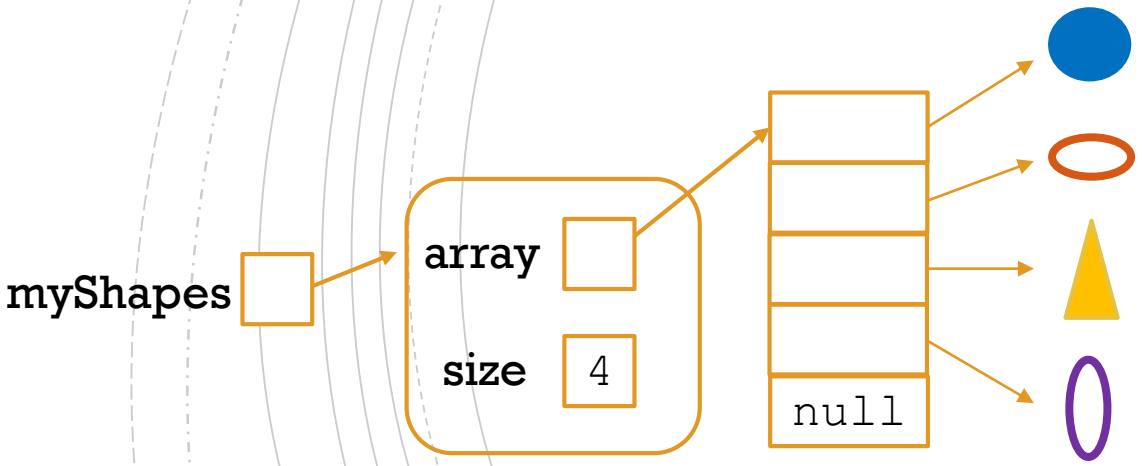
```
ArrayList<Shape> myShapes = … ;
```

What do you think we get if the clone of such `ArrayList`?

```
ArrayList<Shape> copyList = myShapes.clone();
```

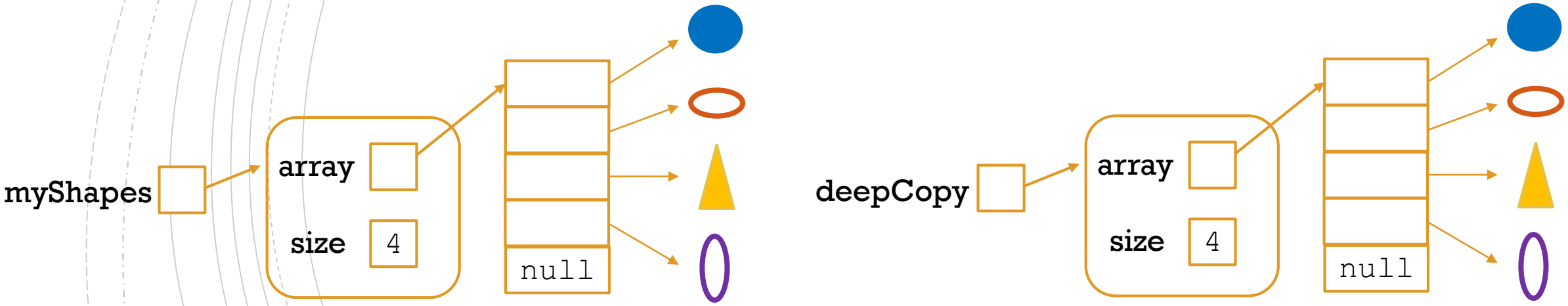
EXAMPLE – HOW TO CLONE AN ARRAYLIST?

```
ArrayList<Shape> myShapes = … ;
```



OPTION 1: "DEEP COPY"

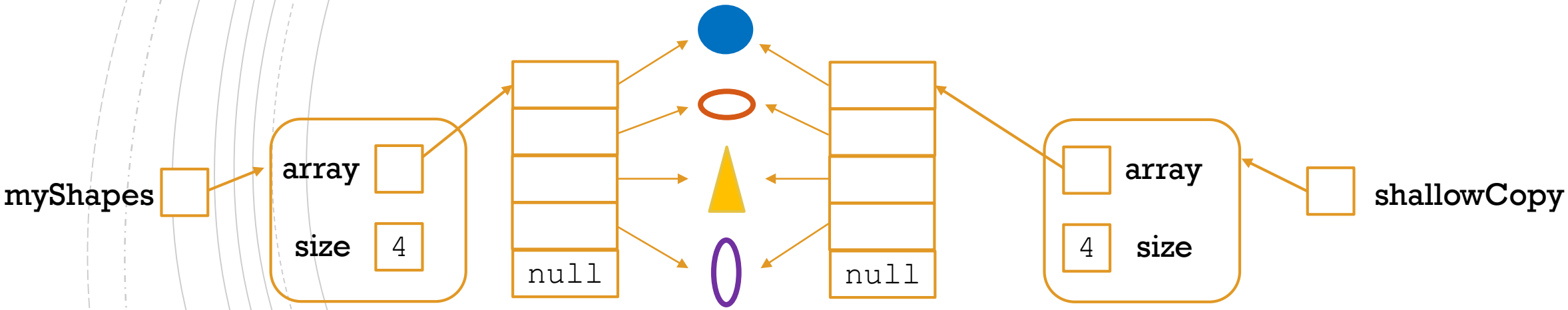
```
ArrayList<Shape> myShapes = … ;
ArrayList<Shape> deepCopy = myShapes.clone();
```



OPTION 2: "SHALLOW COPY"

```
ArrayList<Shape> myShapes = … ;

ArrayList<Shape> shallowCopy = myShapes.clone();
```



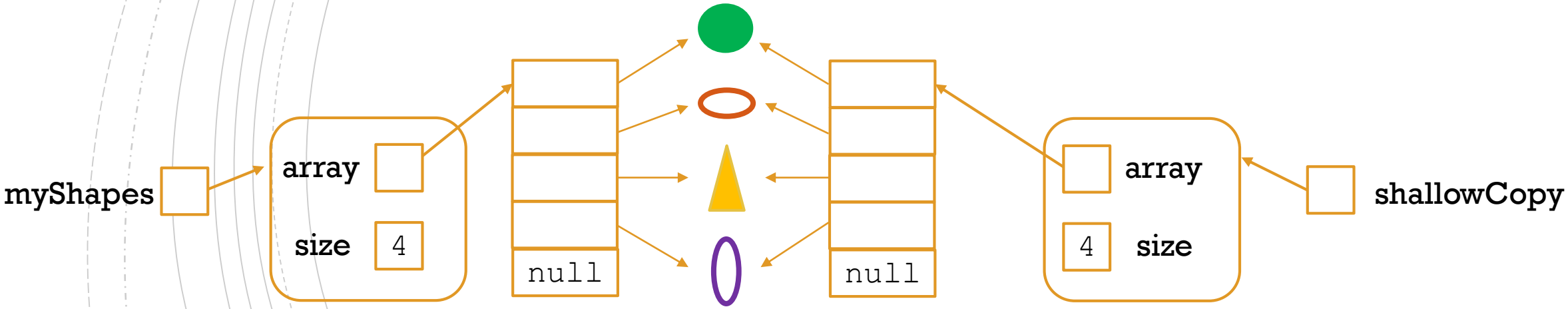
OPTION 2: "SHALLOW COPY"

Note: modifications made to the elements of the list through either `myShapes` or `shallowCopy` can also be seen by the other.

```
ArrayList<Shape> myShapes = … ;

ArrayList<Shape> shallowCopy = myShapes.clone();

shallowCopy.get(0).setColor("green");
```



clone() FROM ArrayList

clone

```
public Object clone()
```

Returns a shallow copy of this `ArrayList` instance. (The elements themselves are not copied.)

Overrides:

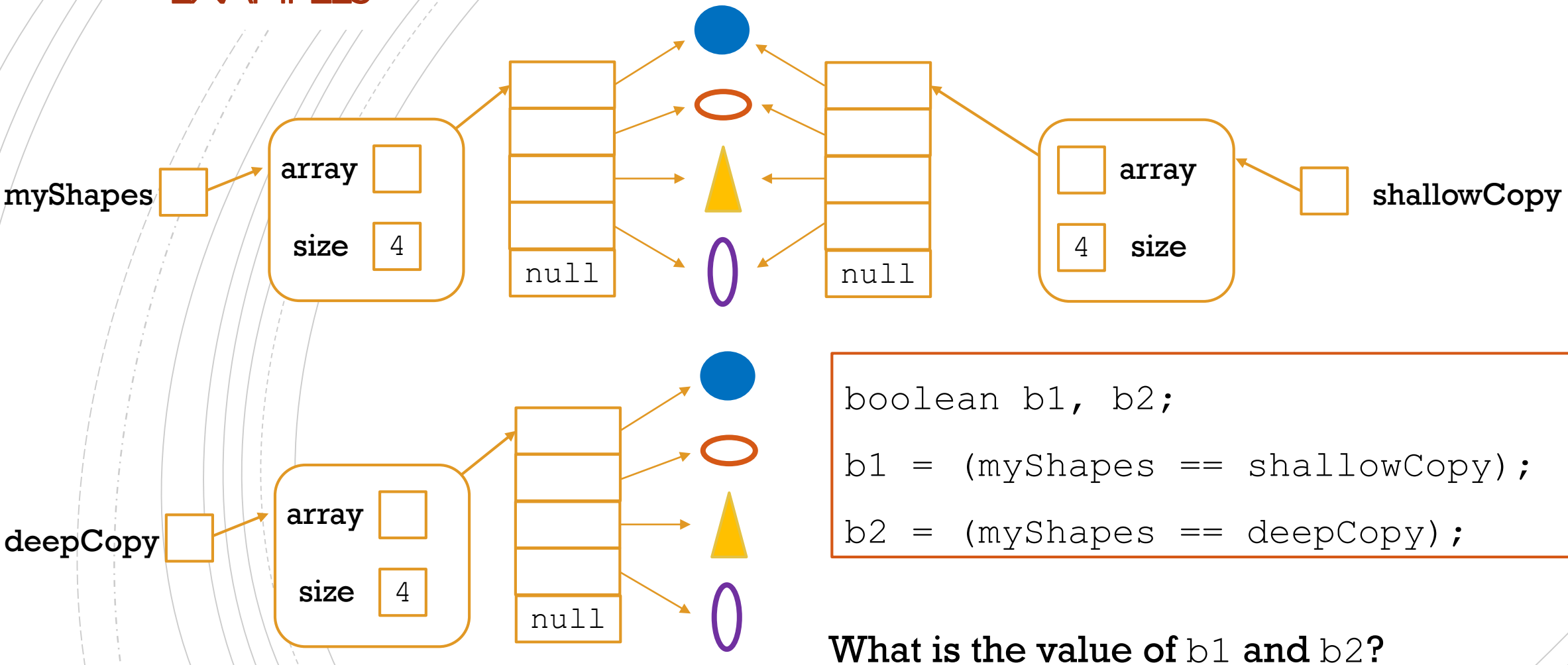
`clone` in class `Object`

Returns:

a clone of this ArrayList instance

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#clone--

EXAMPLES

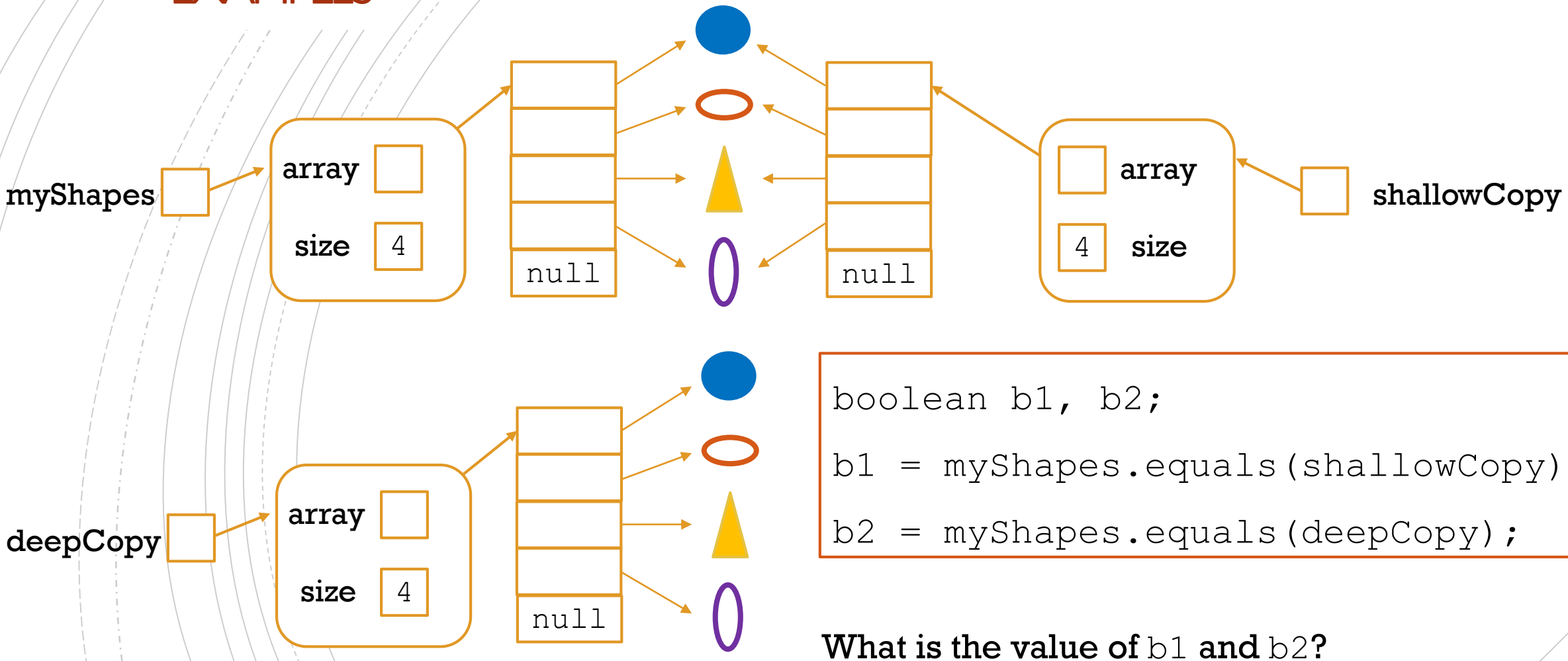



```
boolean b1, b2;
b1 = (myShapes == shallowCopy);
b2 = (myShapes == deepCopy);
```

What is the value of `b1` and `b2`?

- `b1` **is** `false`, `b2` **is** `false`

EXAMPLES




```
boolean b1, b2;
b1 = myShapes.equals(shallowCopy);
b2 = myShapes.equals(deepCopy);
```

What is the value of `b1` and `b2`?

- **it depends on the implementation of `equals()`**

ArrayList inherits equals() FROM List

equals

```
boolean equals(Object o)
```

Compares the specified object with this list for equality. Returns `true` if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are *equal*. (Two elements `e1` and `e2` are *equal* if `(e1==null ? e2==null : e1.equals(e2))`.) In other words, two lists are defined to be equal if they contain the same elements in the same order. This definition ensures that the equals method works properly across different implementations of the `List` interface.

Specified by:

`equals` in interface `Collection<E>`

Overrides:

`equals` in class `Object`

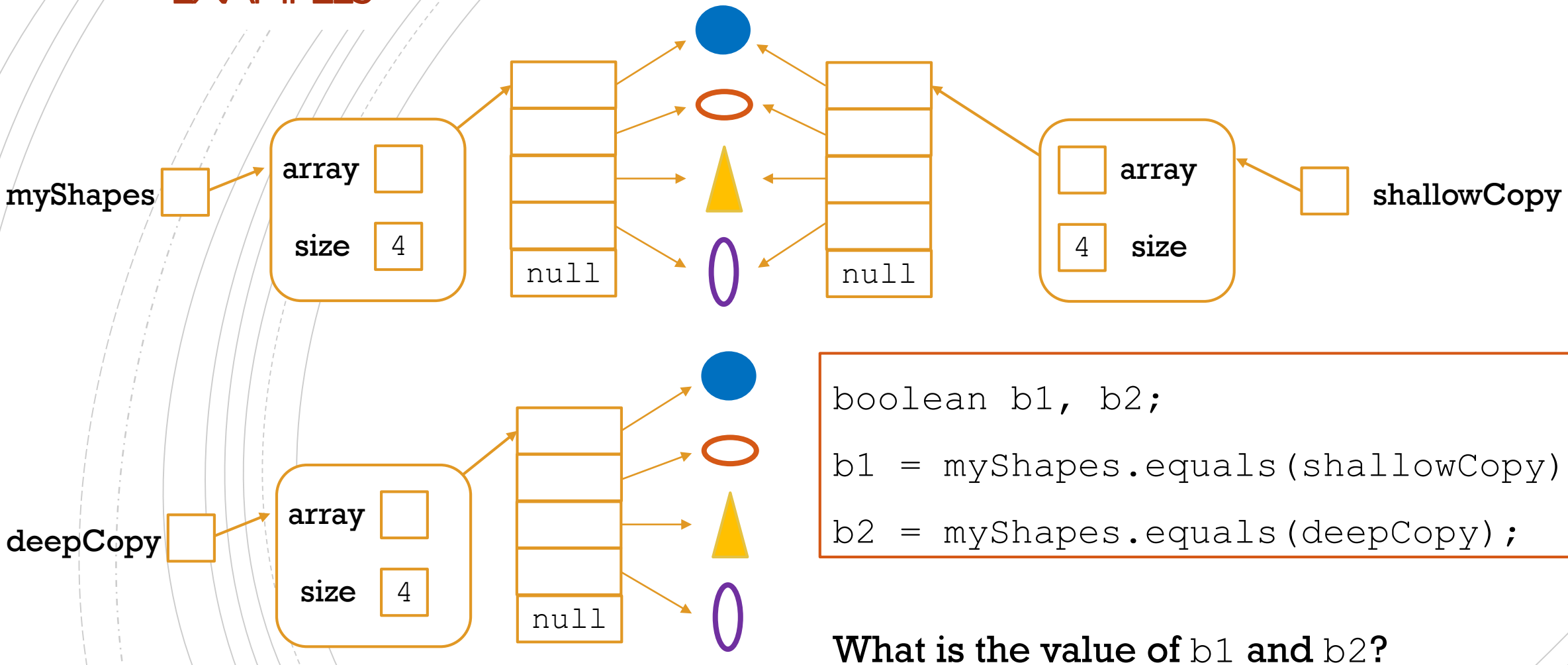
Parameters:

`o` - the object to be compared for equality with this list

Returns:

`true` if the specified object is equal to this list

EXAMPLES




```
boolean b1, b2;
b1 = myShapes.equals(shallowCopy);
b2 = myShapes.equals(deepCopy);
```

What is the value of `b1` and `b2`?

- `b1` **is** `true`, `b2` **is** `false`

TYPE CONVERSION

FROM LAST CLASS

class Dog

Person owner

```
public void bark() {
    print("woof!");
}
    :
```

extends

class Beagle

void hunt ()

```
public void bark() {
    print("aowwwuuu");
}
    :
```

```
public class Test {

    public static void main(String[] args) {

        Dog snoopy = new Beagle();

        snoopy.bark();

    }

}
```

Is this allowed??

OBJECTS TYPE

- We have seen that an object is of the type of the class from which it was instantiated.

- For example, if we write

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

then `myDog` points to an object of type `Dog`.

OBJECT TYPES

- **But `Dog` is a subclass of `Animal` which is a subclass of `Object`.**
- **Thus, a `Dog` is an `Animal` and is also an `Object`. We can use an object of type `Dog` wherever objects of type `Animal` or `Object` are called for.**
- **Note that the reverse is not necessarily true: an `Animal` could be a `Dog`, but not necessarily. Similarly, an `Object` could be an `Animal` or a `Dog`, but it isn't necessarily.**

TYPE CASTING – REFERENCE TYPES

- Casting allows us to use an object of one type in place of another type, if permitted.

- For example we can write

```
Animal myPet = new Dog();
```

This will not cause a compile-time error because there is an ***implicit upcasting*** since a `Dog` is for sure also an `Animal`.

TYPE CASTING – REFERENCE TYPES

On the other hand, consider the following

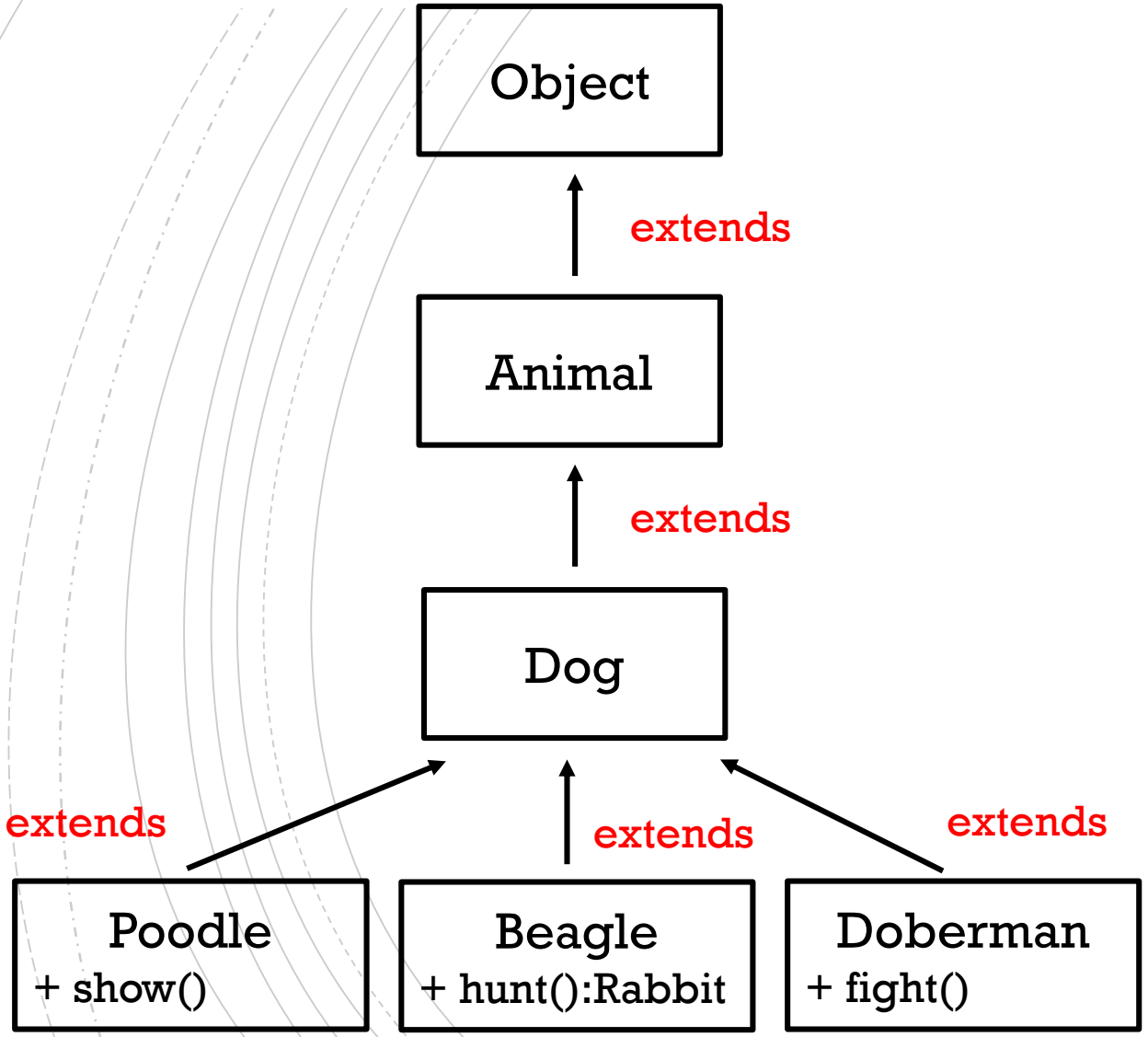
```
Animal myPet = new Dog();
Dog myDog = myPet;
```

The second line will cause a compile-time error. From the compiler point of view, `myPet` is of type `Animal` and an `Animal` might not be a `Dog`.
However, we can tell the compiler that `myPet` is of the correct type, by ***explicitly downcasting***:

```
Dog myDog = (Dog) myPet;
```

If `myPet` turns out to be of the wrong type we'll get a run-time error.

HIERARCHY FROM LAST CLASS



Upcasting

Happens automatically

IMPORTANT!
Note that casting does NOT change the object itself, it just labels it differently!

Downcasting

The programmer has to manually do it.

EXAMPLES

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

Is this allowed?

- Yes, it is an example of upcasting which happens automatically.

EXAMPLES

```
Dog myDog = new Beagle();

Poodle myPoodle = myDog;
```

Is this allowed?

- **Compile-time error! The variable `myDog` is of type `Dog`, and it might not be pointing to a `Poodle`. It requires explicit downcasting to compile.**

EXAMPLES

```
Dog myDog = new Beagle();
Poodle myPoodle = (Poodle) myDog;
```

Is this allowed?

- **The code compiles, but there will be a run-time error because `myDog` is not pointing to a `Poodle` after all.**

EXAMPLES

```
Dog myDog = new Beagle();
myDog.hunt();
```

Is this allowed?

- **Compile-time error! The variable `myDog` is of type `Dog`, and there is no method called `hunt` inside the `Dog` class.**

EXAMPLES

```
Dog myDog = new Beagle();
((Beagle) myDog).hunt();
```

Is this allowed?

- Yes, this code will compile and run.

A LITTLE ABOUT `instanceof`

- **The `instanceof` operator is used to test whether an object is an instance of the specified type.**
- **It returns either `true` or `false`. If we apply the `instanceof` operator with any variable that has `null` value, it returns false.**

```
Dog myDog = new Dog();
Beagle snoopy = new Beagle();
Dog aDog = null;
System.out.println(myDog instanceof Dog); // true
System.out.println(snoopy instanceof Dog); // true
System.out.println(aDog instanceof Dog); // false
```

instanceof AND DOWNCASTING

- **When can use `instanceof` to make sure that downcasting to a subclass will not cause a run time error.**

```
public static void myMethod(Dog myDog) {
   if(myDog instanceof Beagle) {
      Beagle b = (Beagle) myDog; // downcasting
      b.hunt();
   }
}
```

instanceof AND equals()

- Note that in general we will want to use `instanceof` as a last resort. We'll discuss more about this on Monday.
- That said, we have to use `instanceof` when overriding `equals()`

```
public class Dog {
    Person owner;
    :
    public boolean equals(Object obj) {
        if(obj instanceof Dog) {
            ...
        }
    }
}
```

NEXT CLASS!

class Dog

Person owner

```
public void bark() {
    print("woof!");
}
    :
```

extends

class Beagle

void hunt ()

```
public void bark() {
    print("aowwwuuu");
}
    :
```

```
public class Test {

    public static void main(String[] args) {

        Dog snoopy = new Beagle();

        snoopy.bark();

    }

}
```

Which `bark()` will execute???

Is this allowed??

Yes, it's an example of upcasting!