COMP 250

Lecture 13

inheritance 1:
constructor chaining and `super`,
overloading vs. overriding , `final`,
`Object` class: `equals, clone`

Feb. 4, 2022

All dogs are animals.

All beagles are dogs.

relationships
between classes

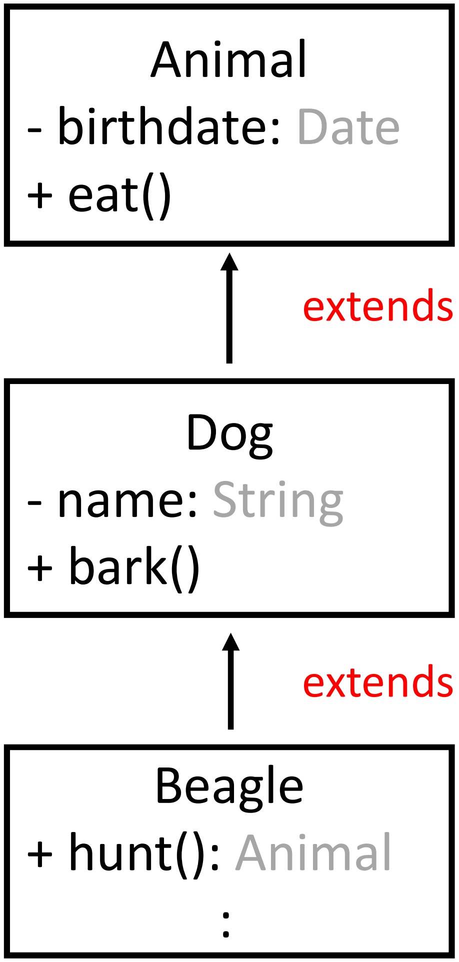
Animals are born, eat food, etc.

Dogs bark (and are born, eat, etc)

Beagles chase rabbits and other animals
(and bark, are born, eat, etc)

class
definitions

We can model such class relationships in Java using inheritance.
We can define subclass/superclass as follows.



Animal
- birthdate: Date
+ eat()

extends

Dog
- name: String
+ bark()

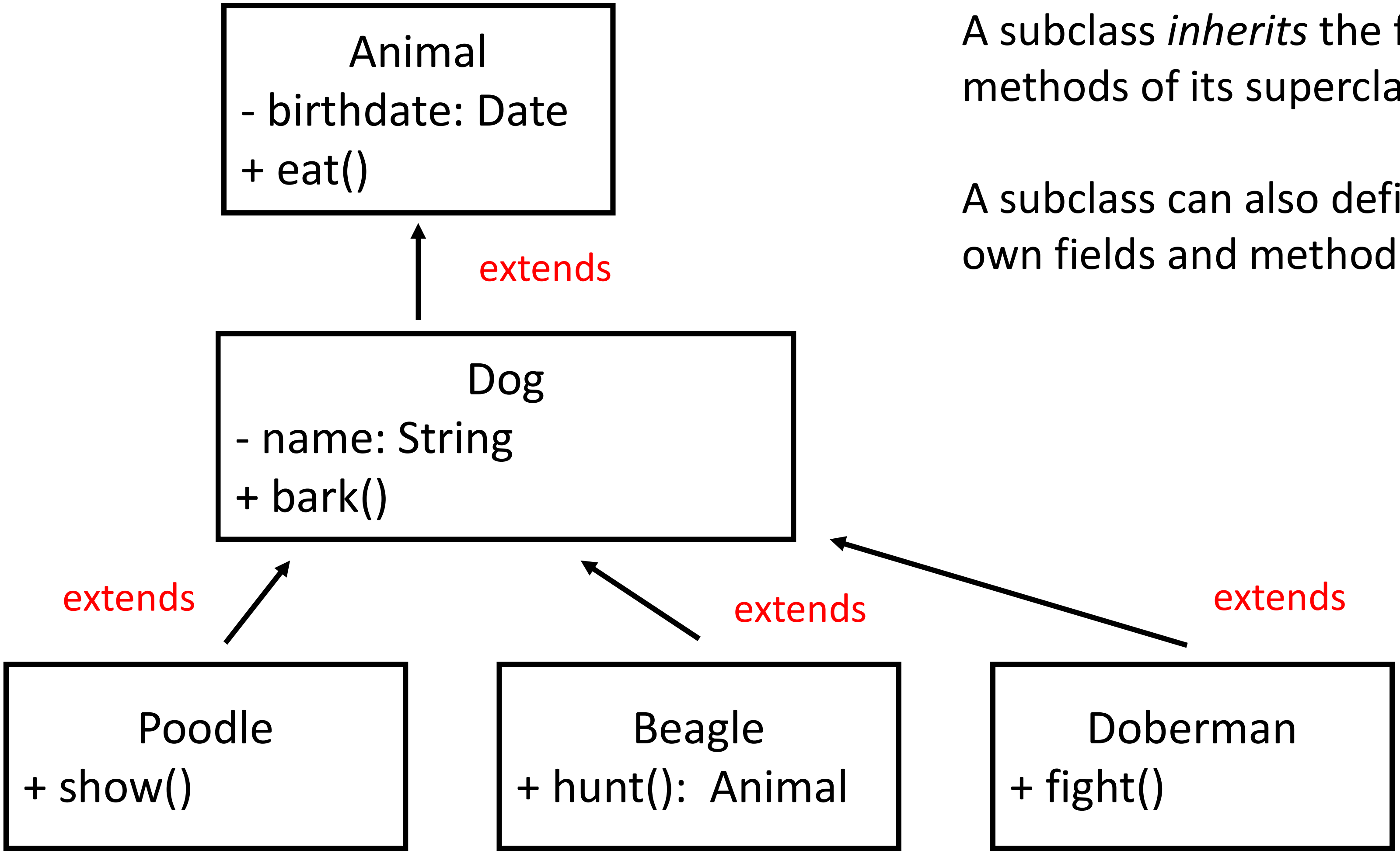
extends

Beagle
+ hunt(): Animal
:

Animal is a *superclass* of Dog.
Dog is a *subclass* of Animal.

Dog is a *superclass* of Beagle.
Beagle is a *subclass* of Dog.

Inheritance in Java



Animal
- birthdate: Date
+ eat()

extends

Dog
- name: String
+ bark()

extends

extends

extends

Poodle
+ show()

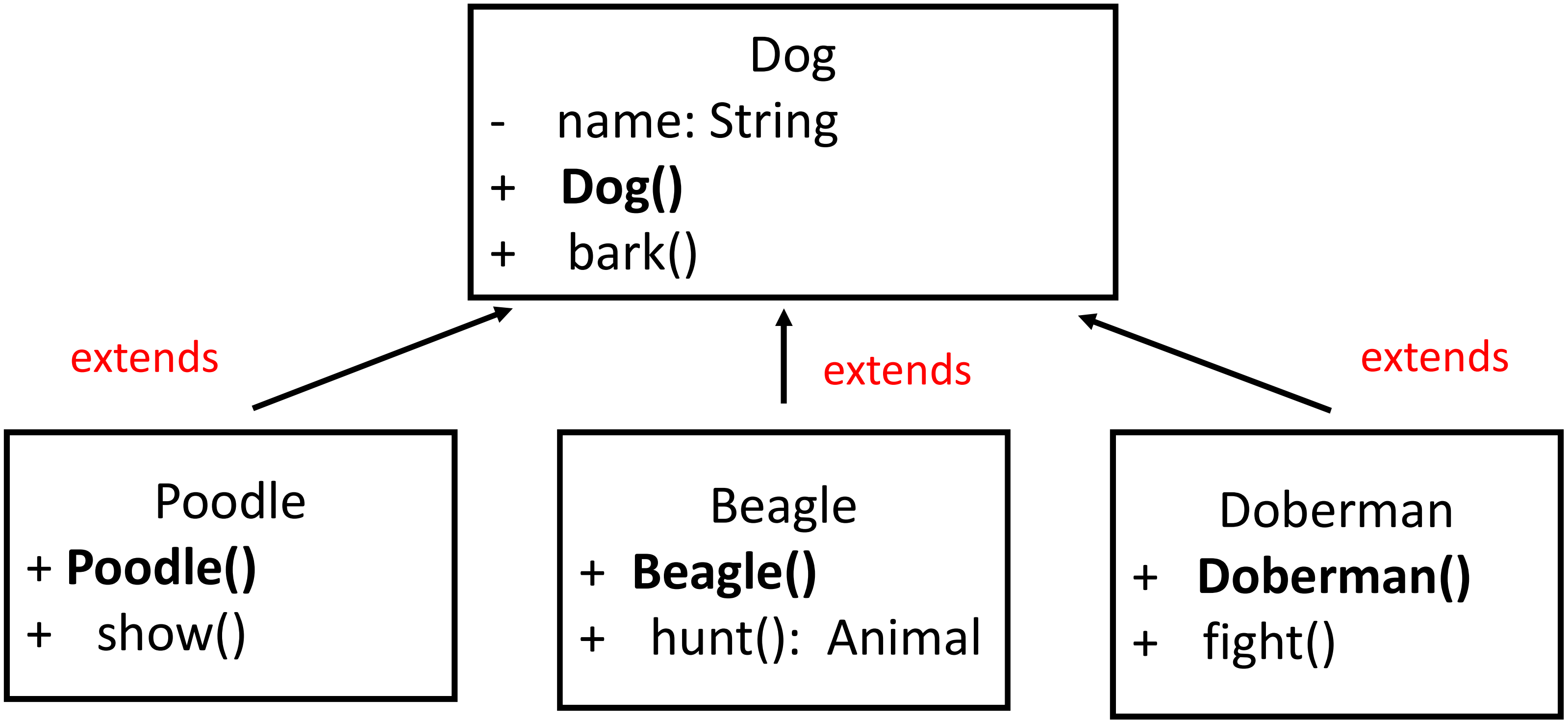
Beagle
+ hunt(): Animal

Doberman
+ fight()

A subclass *inherits* the fields and methods of its superclass.

A subclass can also define its own fields and methods.

Each class has a constructor



Constructor methods are not inherited.

Q: Why not?

A: Each object is an instance of a unique class.

```
class Animal {


   Animal() {

   }
}


class  Dog  extends  Animal {

   Dog() {  }
}
```

Constructor chaining

```
class Animal {
  Date   birthdate;

  Animal() {
    this.birthdate  =  new Date();
  }
}


class  Dog  extends  Animal {

  Dog() {  }

}
```

The no-argument constructor of Dog *automatically* calls the superclass'es no-argument constructor.

The superclass'es constructor (Animal) creates and initializes the fields that are inherited by the subclass (Dog).

Constructor chaining (keyword super)

```
class Animal {
  Date   birthdate;
  String  birthplace;

  Animal() {
    this.birthdate  =  new Date();
  }

  Animal(String birthplace ) {
    this();
    this.birthplace = birthplace;
  }
}
```

another use of the keyword `this`
(it calls the no argument constructor)

```
class  Dog  extends  Animal {
  String name;

  Dog() {  }    //   automatically calls super().

  Dog(String birthplace,  String  name) {
    super(birthplace);
    this.name  = name;
  }
}
```

ASIDE: For more info on
the use of super, see here.

When we write a constructor with arguments, we can initialize fields inherited from the super class by explicitly calling a superclass constructor.

COMP 250

Lecture 13

inheritance 1:
constructor chaining and `super`,
overloading vs. overriding , `final`,
`Object` class: `equals, clone`

Feb. 4, 2022

As we have just seen...

A subclass can define its own fields.
A subclass can also define fields with the same name as fields in the superclass (called “hiding” the field), but this is considered bad practice because it easily leads to confusion.

A subclass can define its own methods, e.g.

- a method not defined in superclass
- “overloading” a superclass method
- “overriding” a superclass method (to be defined soon)
- “hiding” a superclass method – static only
  (ASIDE: I will skip this one – too many details to think about.)

Method “Signature”

The signature of a method is the name and the parameter list, but *not* the return type or modifiers.

e.g.

```
         double   distanceTo( Point2D  p)
static   double   distanceBetween(Point2D p, Point2D q)
```

Recall: overloading a method

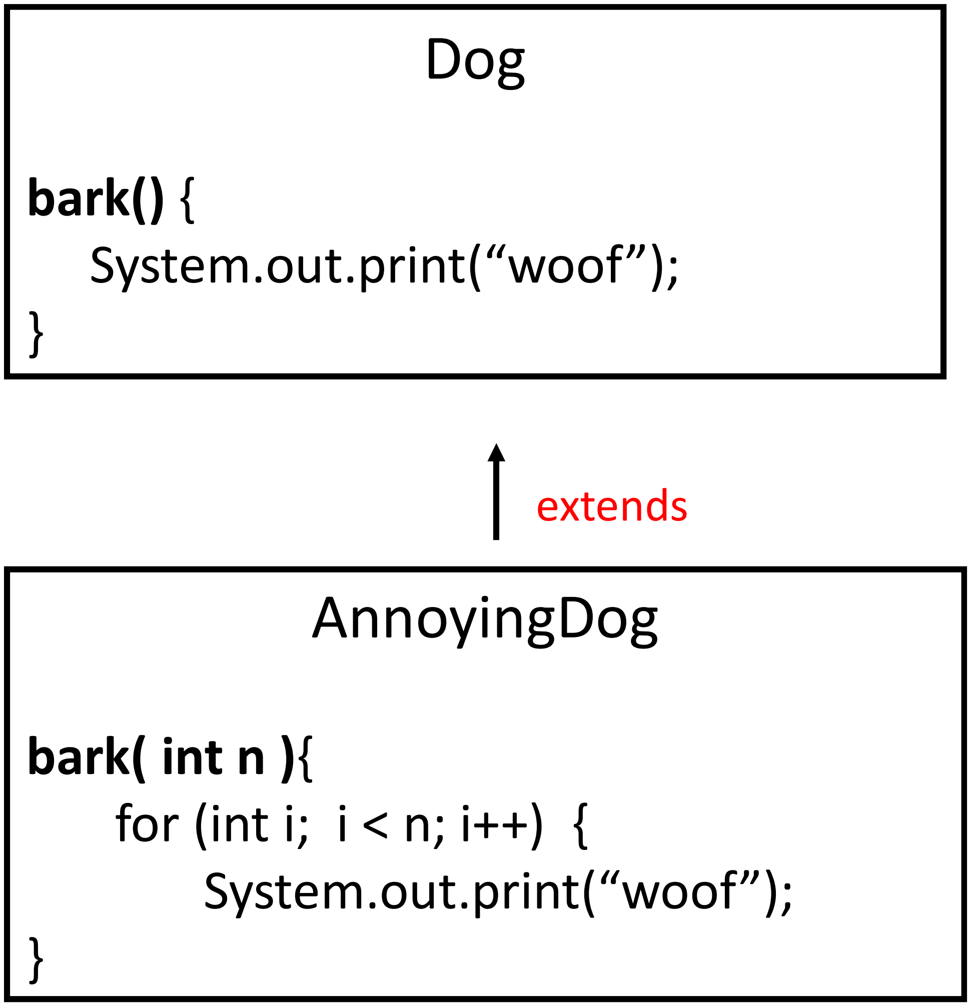
Two or methods can have the same name but different parameter types.

e.g   `LinkedList<E>`

```
void  add( E  e)              //  at tail
void  add( int index, E  e )
```

In this example, overloading occurs for different methods *within* a class.

Overloading *between* classes



Dog

```
bark() {
    System.out.print(“woof”);
}
```

extends

AnnoyingDog

```
bark( int n ){
      for (int i;  i < n; i++)  {
            System.out.print(“woof”);
}
```

Note that the bark() method is inherited. So bark is overloaded in AnnoyingDog.

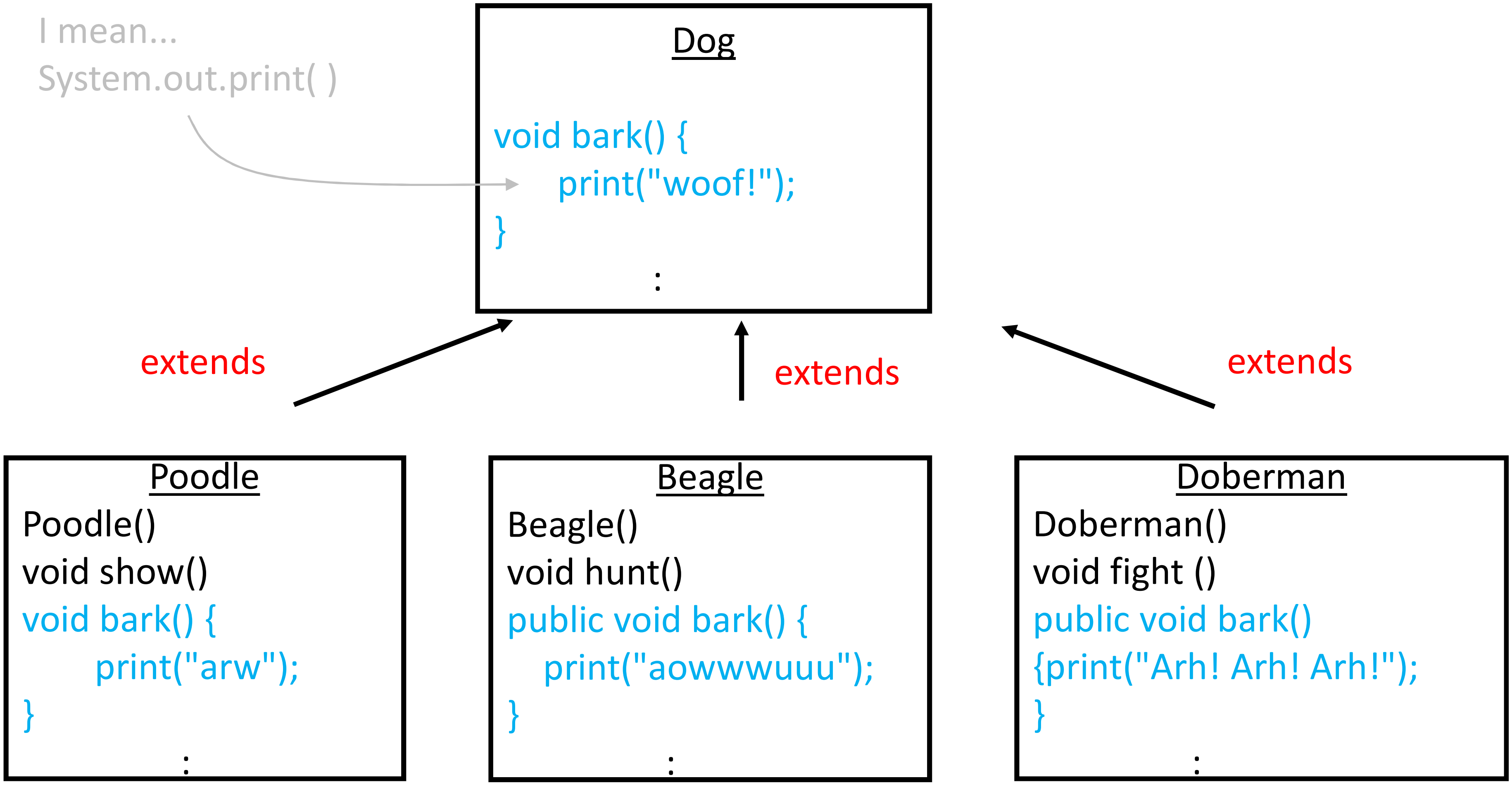
Overriding a method

- *same* method signatures

  (same method name and parameter types)

- always between classes, specifically a subclass method *overrides* a superclass method

Overriding e.g. bark()



I mean...
System.out.print( )

Dog

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

extends

extends

extends

Poodle

```
Poodle()
void show()
void bark() {
      print("arw");
}
        :
```

Beagle

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

Doberman

```
Doberman()
void fight ()
public void bark()
{print("Arh! Arh! Arh!");
}
        :
```

https://www.youtube.com/watch?v=_wqK15EtCMo

https://www.youtube.com/watch?v=esjec0JWEXU

https://www.youtube.com/watch?v=s5Y-Gyt57Dw

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

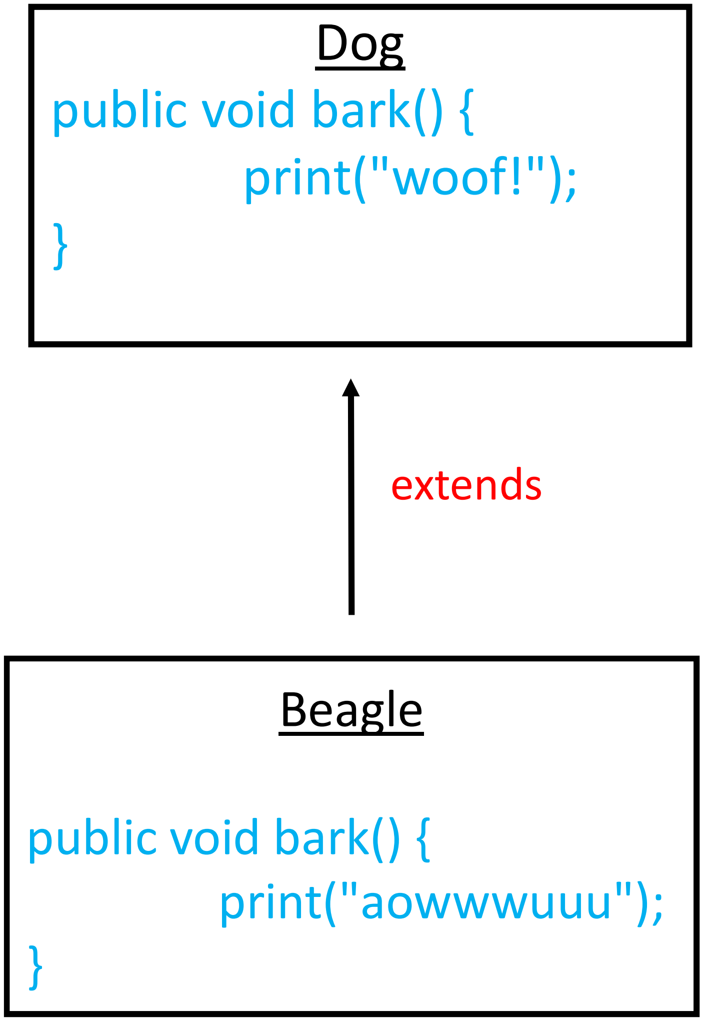
What is printed ?

- ➢ woof!

```
Beagle myDog = new Beagle();
       myDog.bark();
```

What is printed?

- ➢ aowwwuuu



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

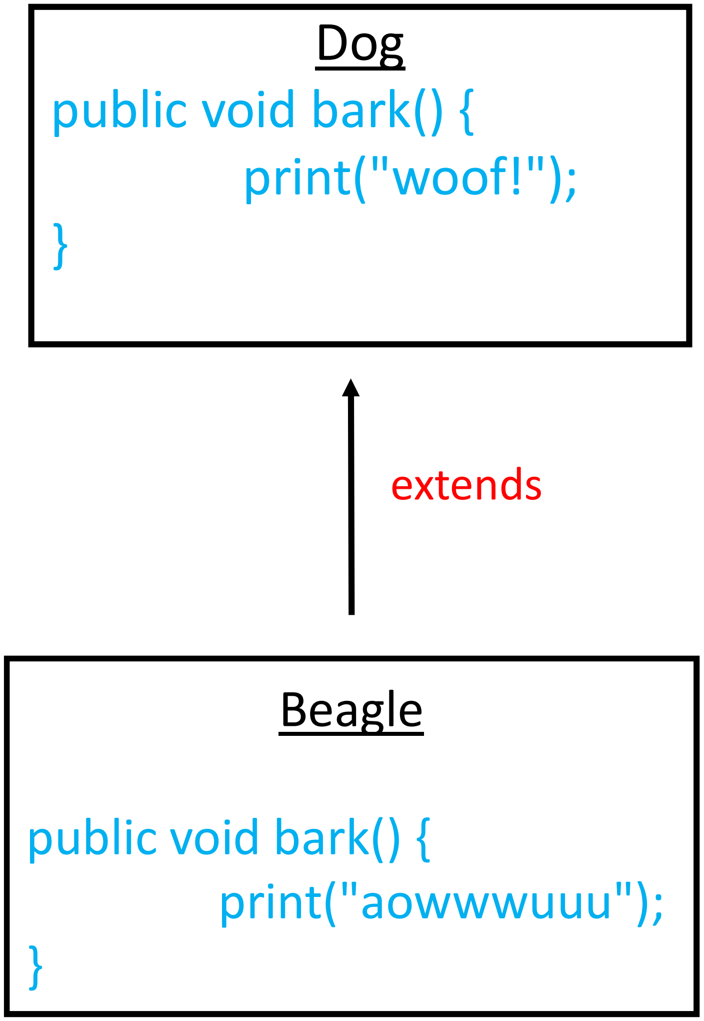
```
public void bark() {
        print("aowwwuuu");
}
```

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

The first line above is allowed!

What is printed ?

We’ll cover this case next lecture.



Dog

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

extends

Beagle

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

`final` keyword

A class that is declared `final` cannot be *extended*.

```
public final class Dog  {
           :
}
```

```
public class Beagle extends Dog {
           :
}
```

✗ compile-time error!

e.g. `Integer, Double, Math, String,` are `final`

`final` keyword

A method that is declared `final` cannot be *overridden*.

```
public class Dog {

   public final void bark() {
                :
        }
}
```

```
public class Beagle extends Dog {

        public void bark() {
                :
        }
}
```

✗ compile-time error!

`final` variable

If a variable is declared to be `final`, its value can ***never*** be changed *after* is has been initialized. *(This definition has nothing to do with inheritance, and I should have mentioned it a few weeks ago.)*

```
final int x = 3;
x = 10;
```

✗ compile-time error!

```
final Dog myDog = new Dog(“Willie");
          myDog = new Dog(“Max");
```

✗ compile-time error!

However, you *can* still change the fields of the object that `myDog` references.

```
final Dog yourDog = new Dog(“Snoopy");
          yourDog.setName(“Max");
```

✓

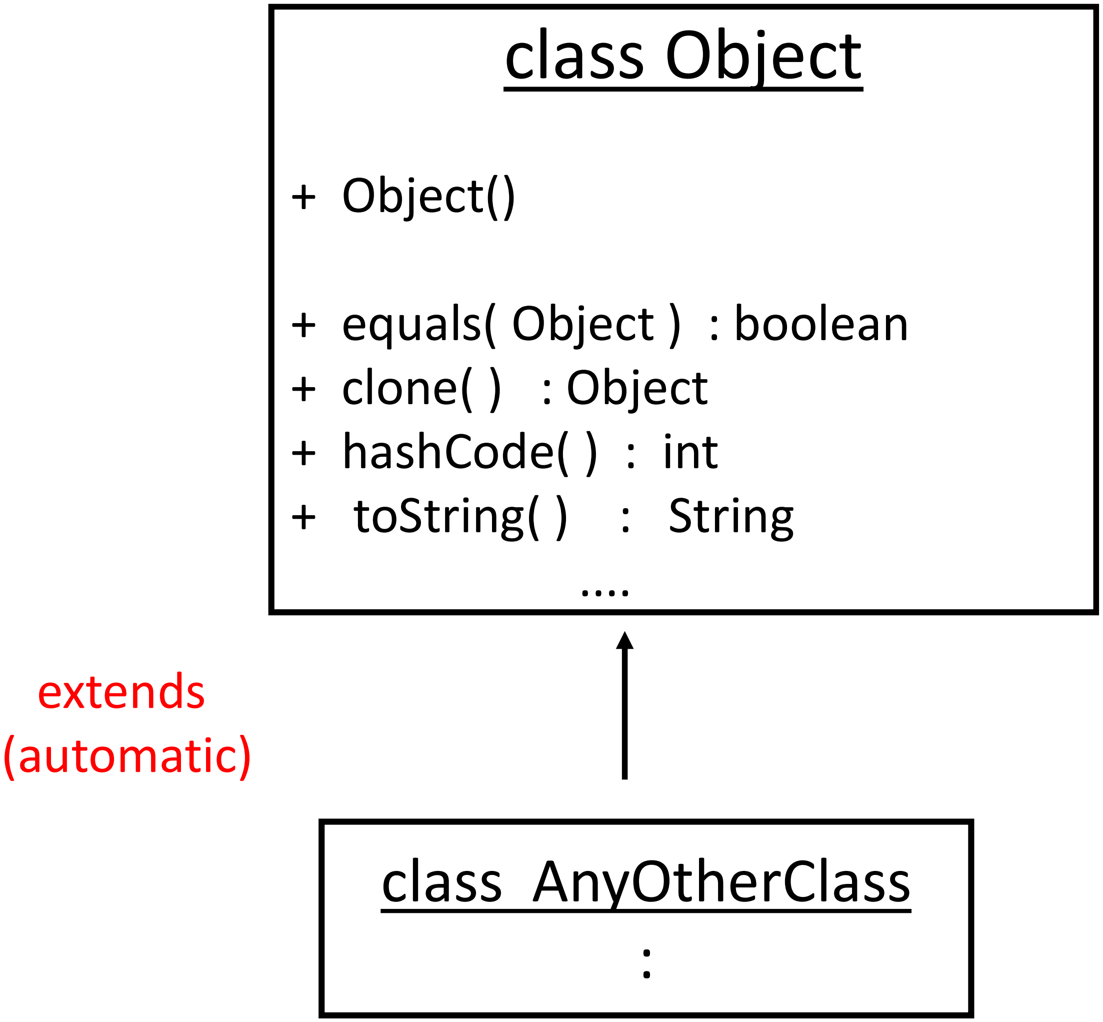
COMP 250

Lecture 13

inheritance 1:
constructor chaining and `super`,
overloading vs. overriding , `final`,
`Object` class: `equals, clone`

Feb. 4, 2022

The `Object` class



class Object

+ Object()

+ equals(Object) : boolean

+ clone( ) : Object

+ hashCode( ) : int

+ toString( ) : String

....

extends

(automatic)

class AnyOtherClass

:

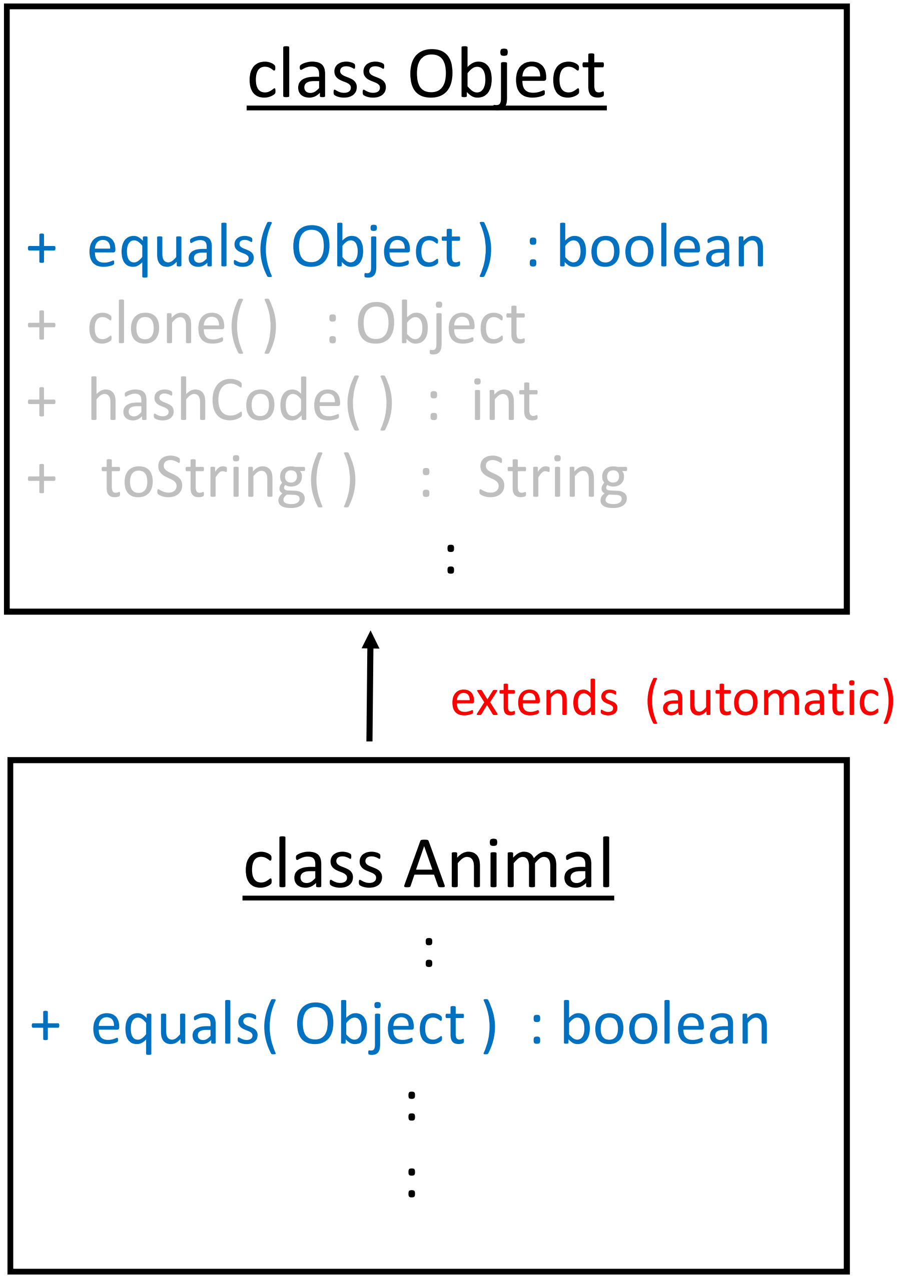
Object.equals(Object o)

This method returns a boolean, saying whether the two arguments are the same object.

```
Object  obj1 = new Object();
Object  obj2 = new Object();
         :
```

`obj1.equals( obj2 )` is equivalent to `obj1 == obj2`

In this example, the result would be `false`.



class Object

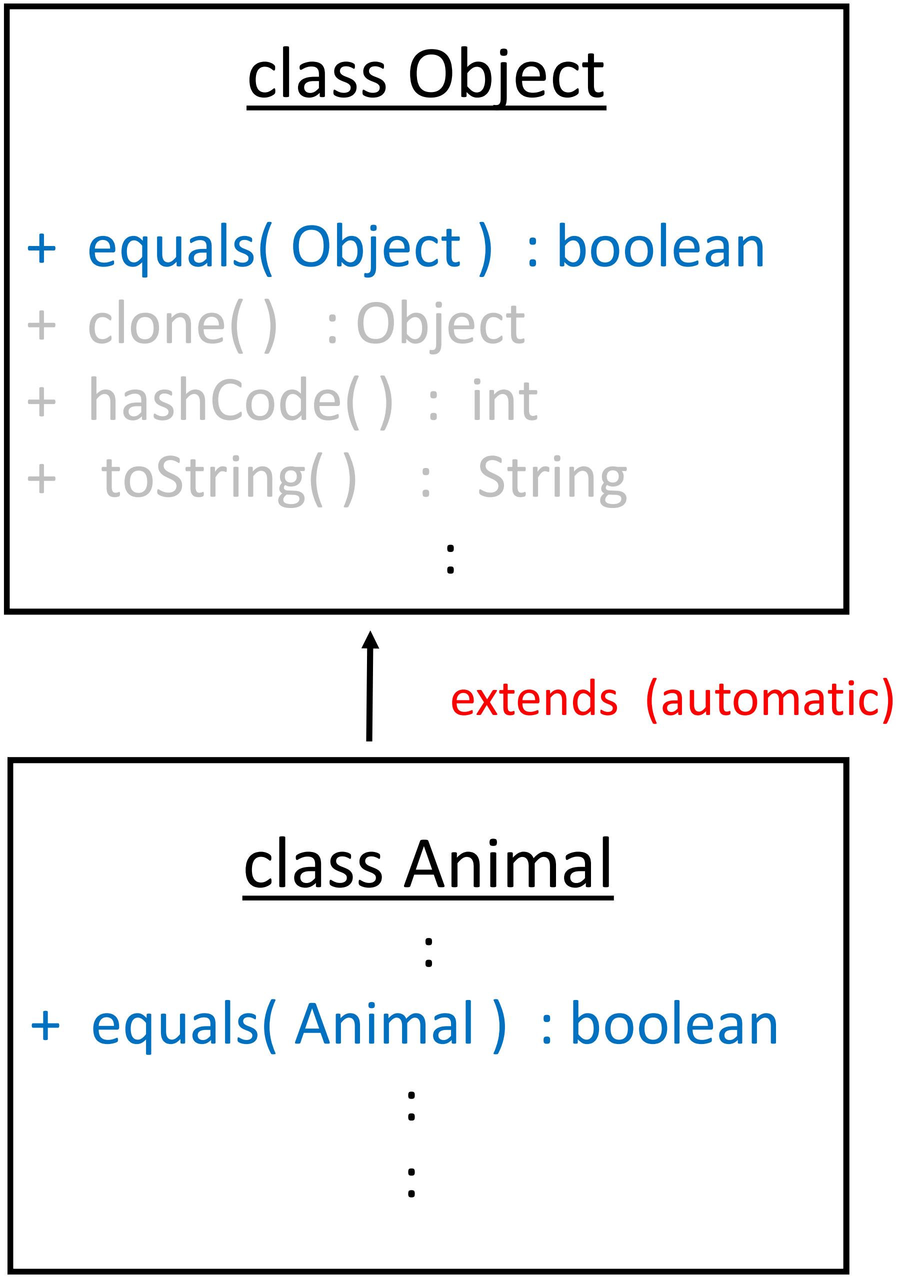
+ equals(Object) : boolean
+ clone( ) : Object
+ hashCode( ) : int
+ toString( ) : String
:

extends (automatic)

class Animal
:
+ equals(Object) : boolean
:
:

Animal.equals(Object) would *override* Object.equals(Object).

We will discuss this more next week.



class Object

+ equals(Object) : boolean

+ clone( ) : Object

+ hashCode( ) : int

+ toString( ) : String

:

extends (automatic)

class Animal

:

+ equals(Animal) : boolean

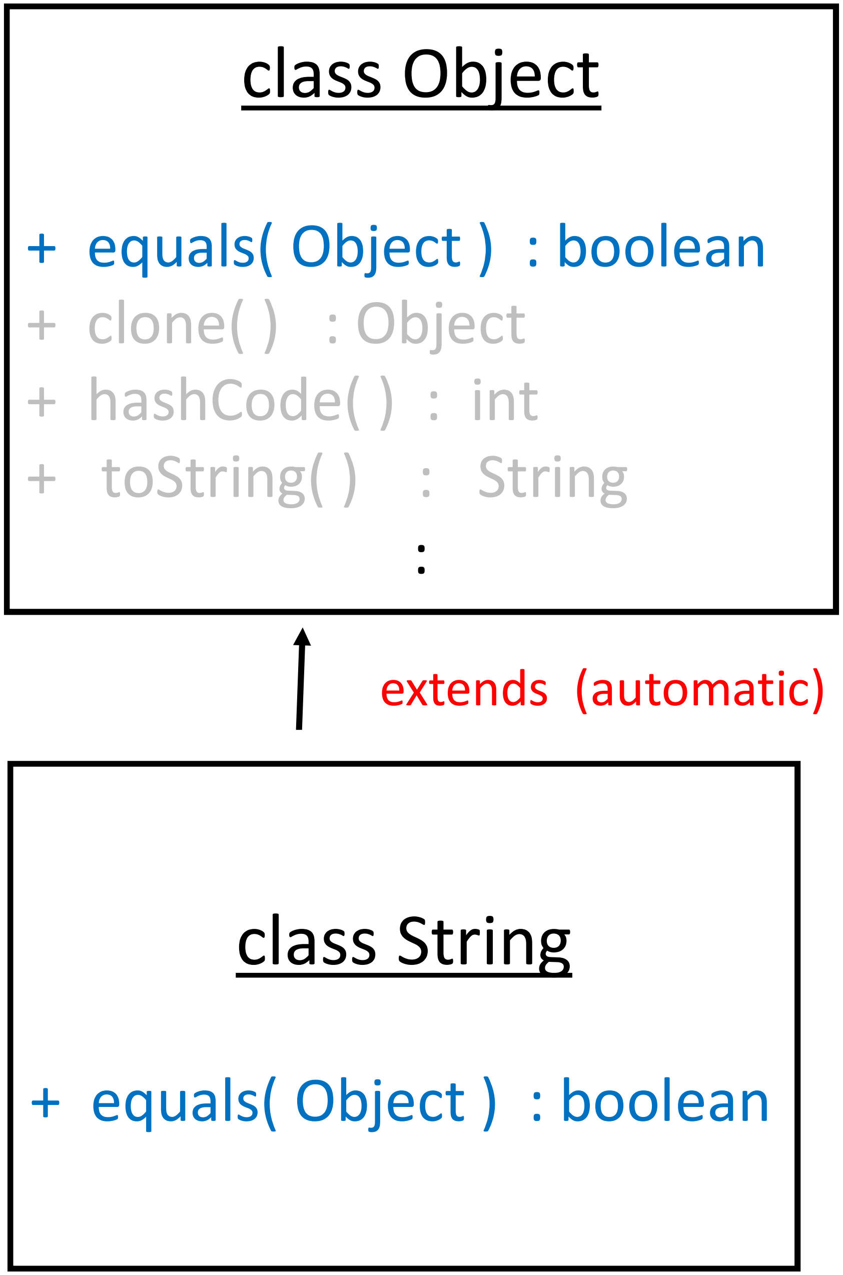
:

:

In this example, the Animal.equals method would be *overloaded*.

Both Animal.equals(Animal) and Animal.equals(Object) would exist in the Animal class.

This is allowed, but it is strongly ***not recommended*** because it creates confusion about which method is called.



class Object

+ equals(Object) : boolean

+ clone( ) : Object

+ hashCode( ) : int

+ toString( ) : String

:

extends (automatic)

class String

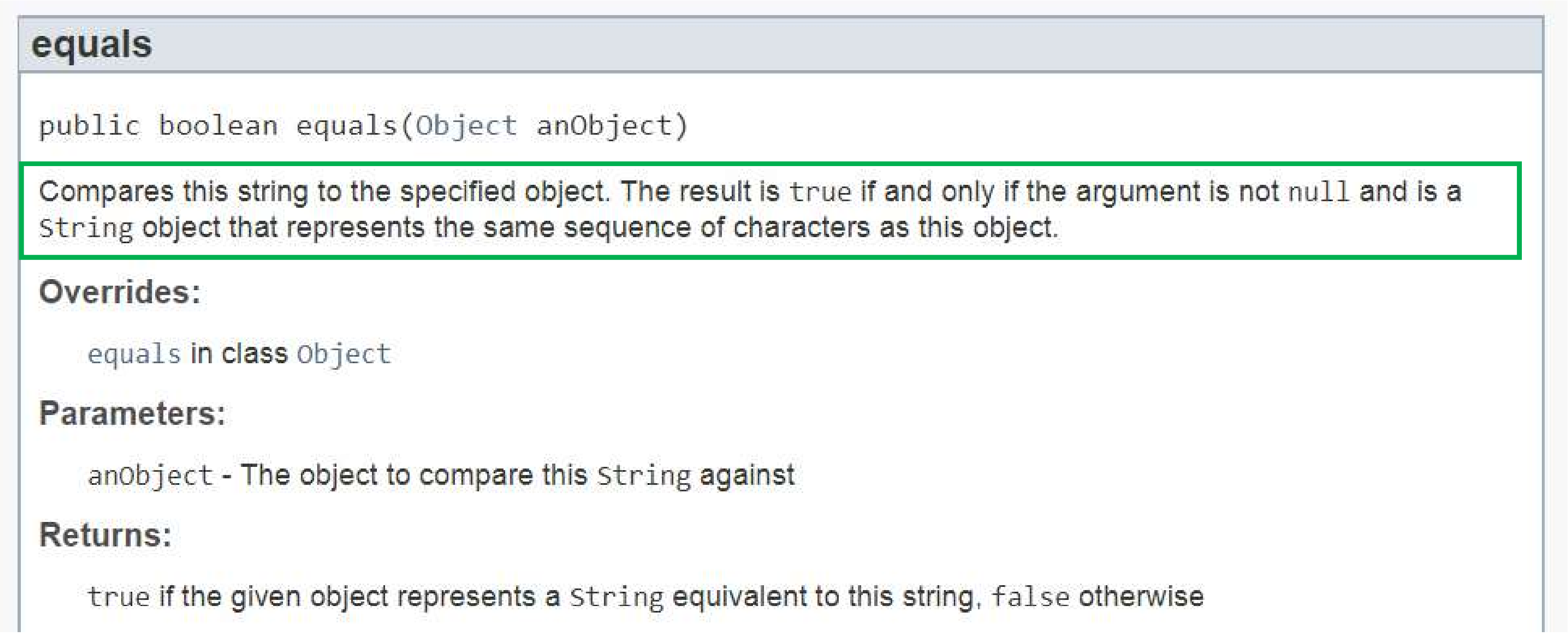
+ equals(Object) : boolean

String.equals(Object)
overrides
Object.equals(Object)

Recall how String.equals()
is defined (next slide).

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

String.equals(Object)



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`

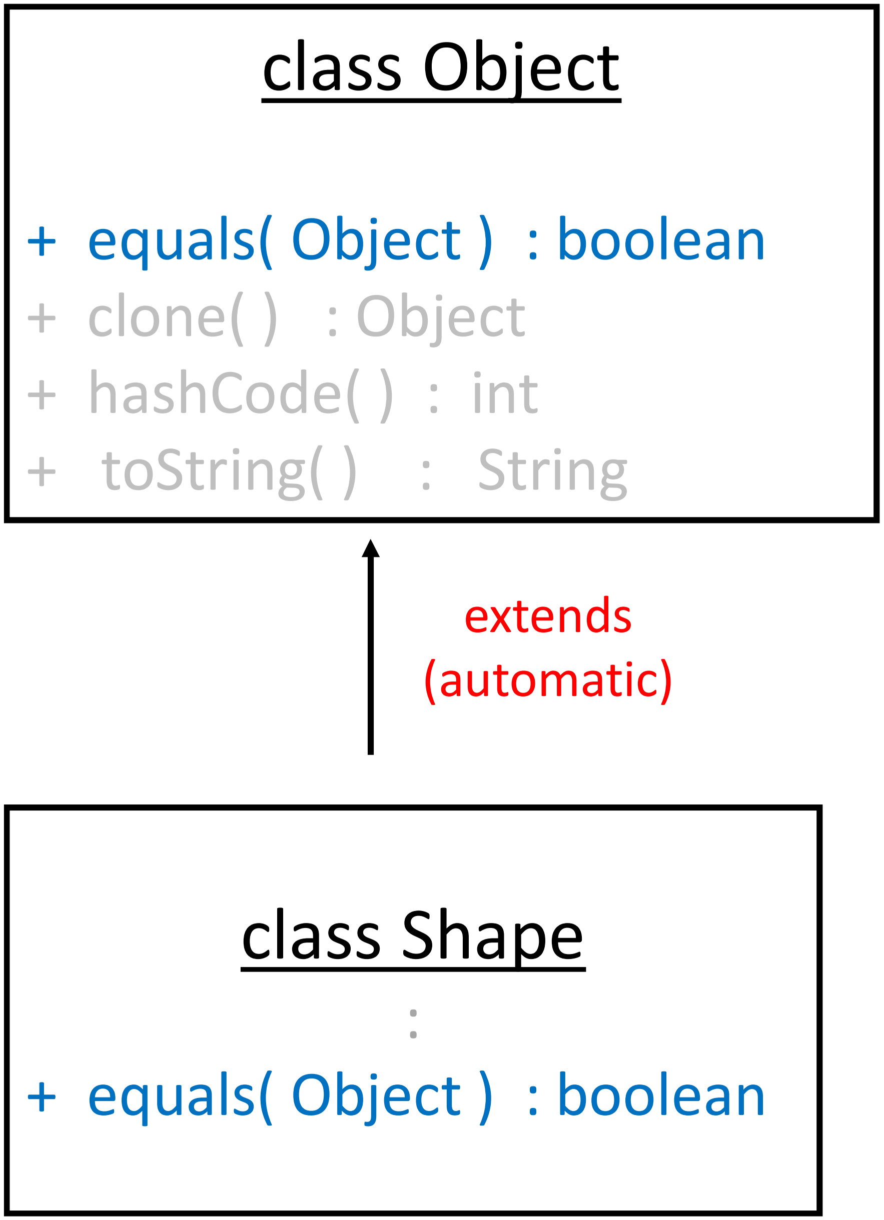
Parameters:

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

Returns:

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

Recall our discussion from lecture 6 how you should use the equals method to compare strings, rather than ==.



class Object

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

extends (automatic)

class Shape

:

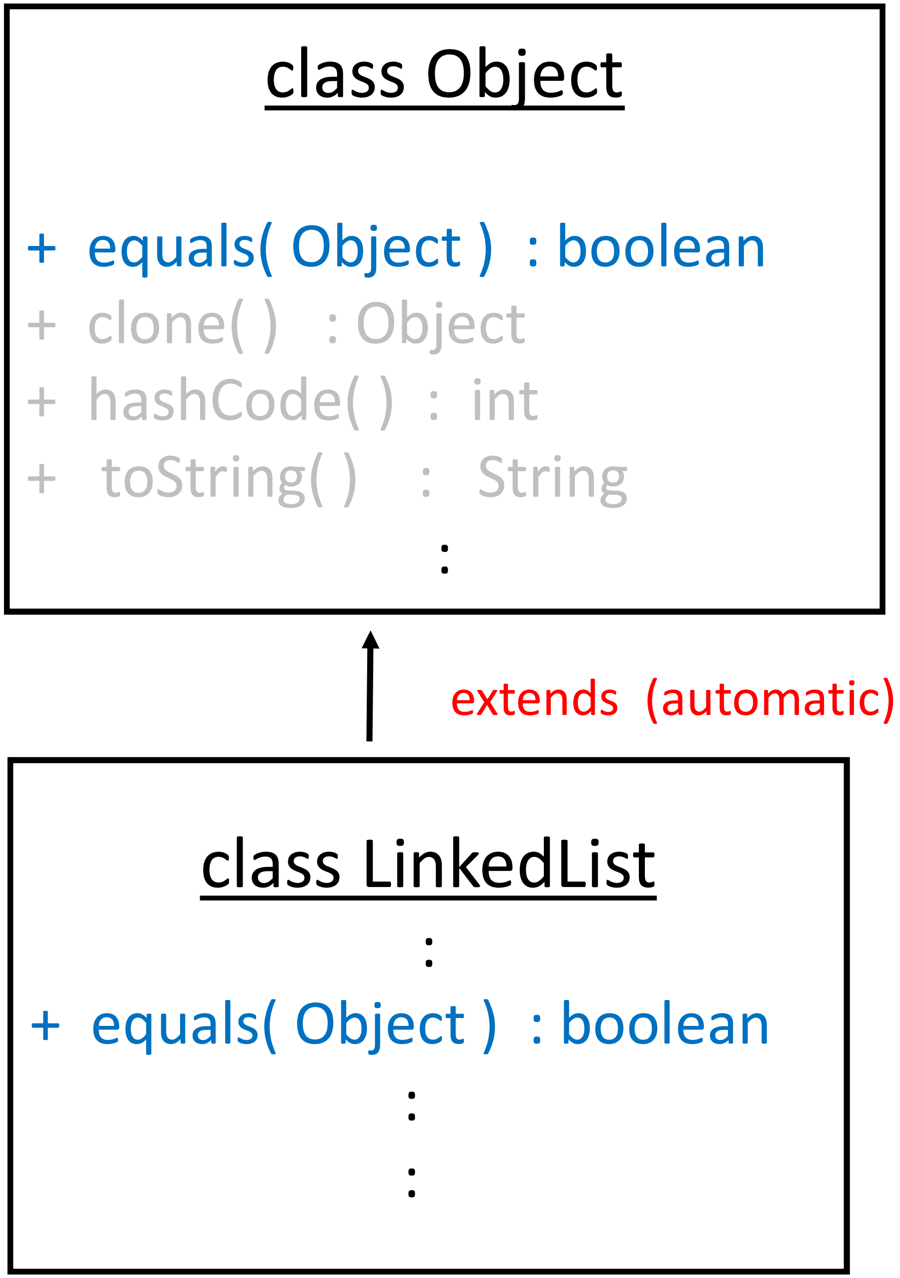
+ equals(Object) : boolean

Let Shape.equals(Object) override Object.equals(Object).

How to define it?
Q: When should two Shape objects be equal ?

s1.equals(s2) returns what ?

A: There is no “right answer” to this question. It depends what you want to achieve.



class Object

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

extends (automatic)

class LinkedList
:
+ equals(Object) : boolean
:
:

LinkedList.equals(Object)
overrides
Object.equals(Object)

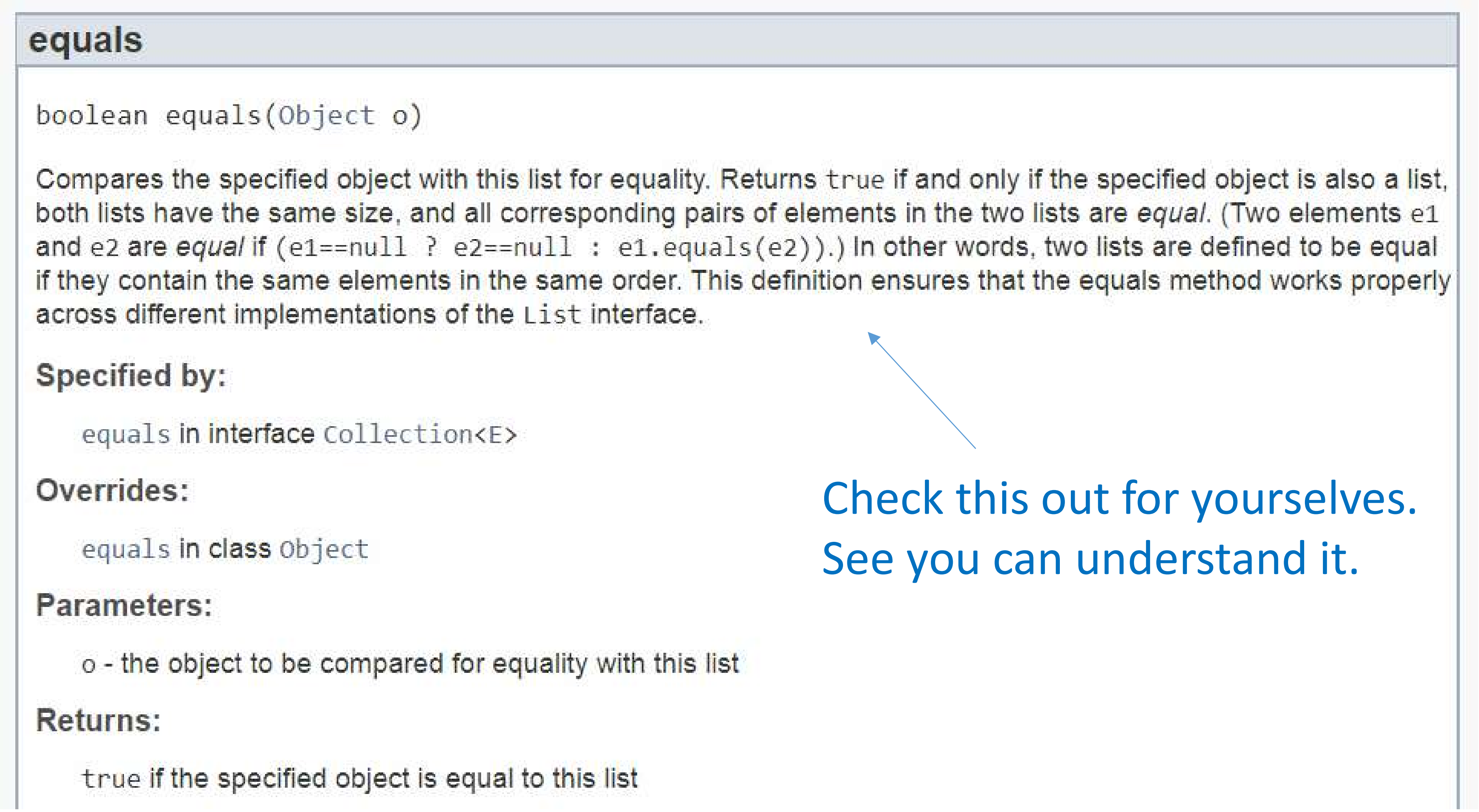
Q: When are two LinkedList objects equal ?

A: It depends...

For example, two LinkedList<Shape> objects are equal if and only if the sizes of the lists are the same *and the* Shape *objects stored at corresponding nodes are themselves equal,* according to the Shape class'es equals() method.

https://docs.oracle.com/javase/7/docs/api/java/util/List.html

LinkedList.equals(Object)



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:

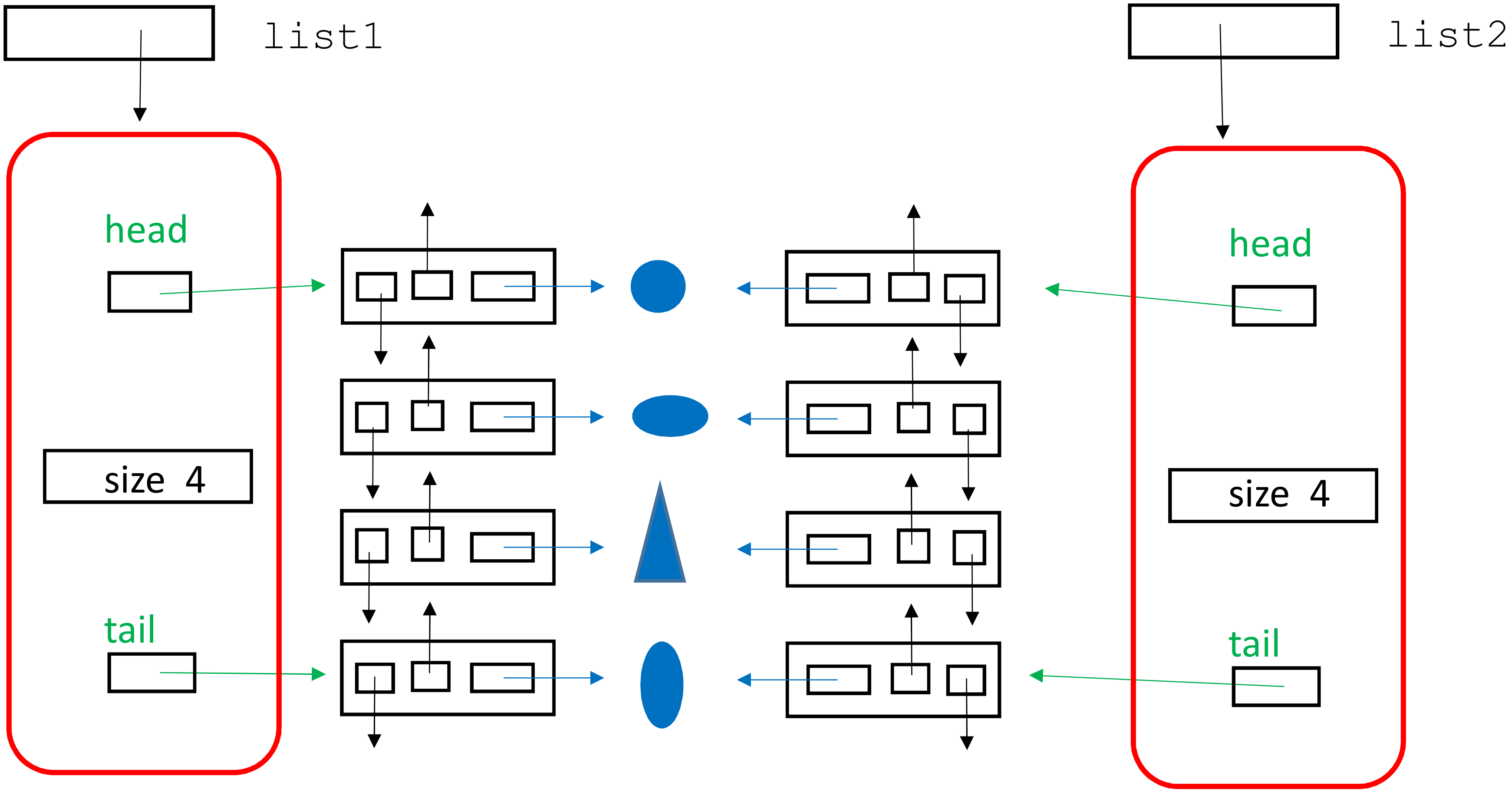
`o` - the object to be compared for equality with this list

Returns:

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

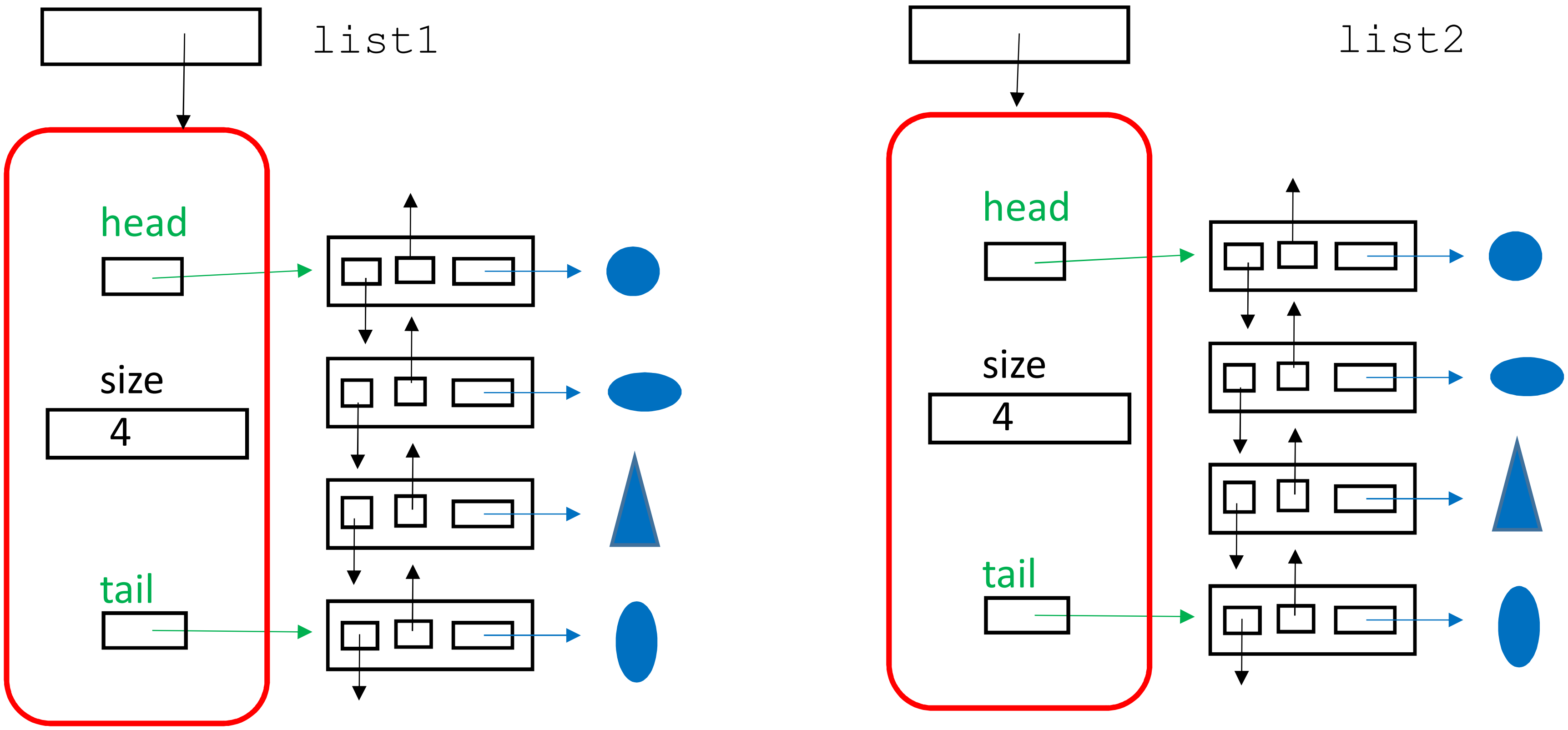
Check this out for yourselves.
See you can understand it.

“Shallow copy” (recall lecture 11)



`list1.equals(list2)` is `true.`   Why ?

“Deep copy” (recall lecture 11)



Q: `list1.equals(list2)` == `?`

A: It depends how `Shape.equals(Object)` is defined.

Object.equals(Object)

`Object  obj1, obj2`

`obj1.equals( obj2 )` is equivalent to `obj1 ==  obj2`

Q: Are there any general rules/guidelines for how a class *should* override `Object.equals( Object )` ?

A: Yes !

Rules for `equals( Object )`

`x.equals(x)` should always return true

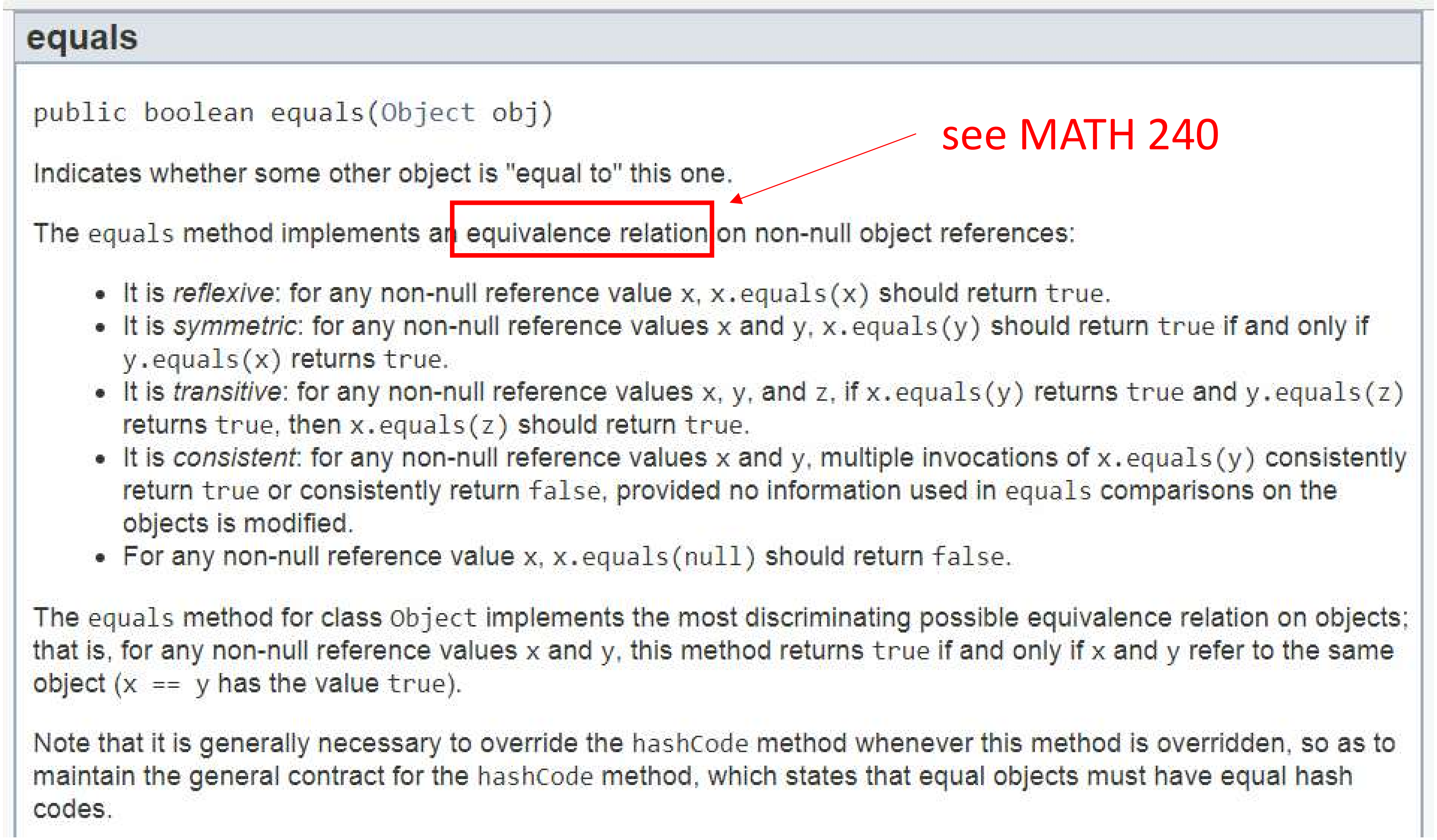
`x.equals(y)` should return true if and only if
`y.equals(x)` returns true

if `x.equals(y)` and `y.equals(z)` both return true, then
`x.equals(z)` should return true

`x.equals(null)` should return false when `x` references any object.

… and more

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



equals

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

see MATH 240

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

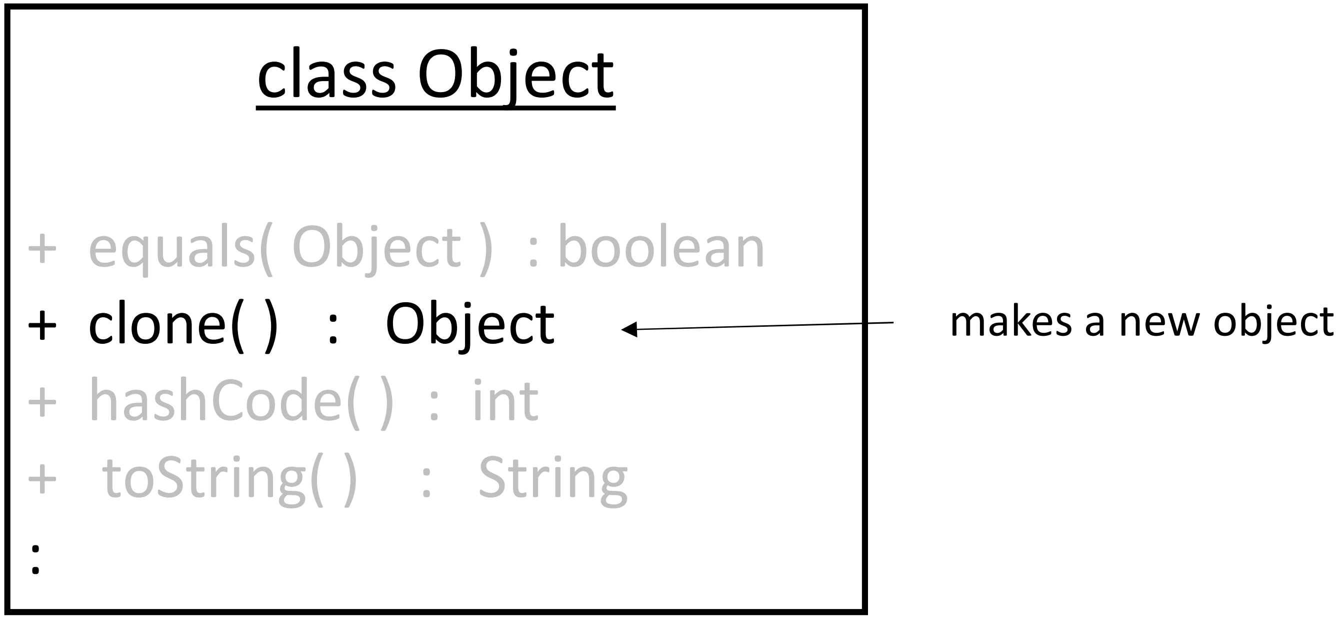
The `equals` method implements an equivalence relation on non-null object references:

- 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.

Object.clone()



class Object

+ equals(Object) : boolean

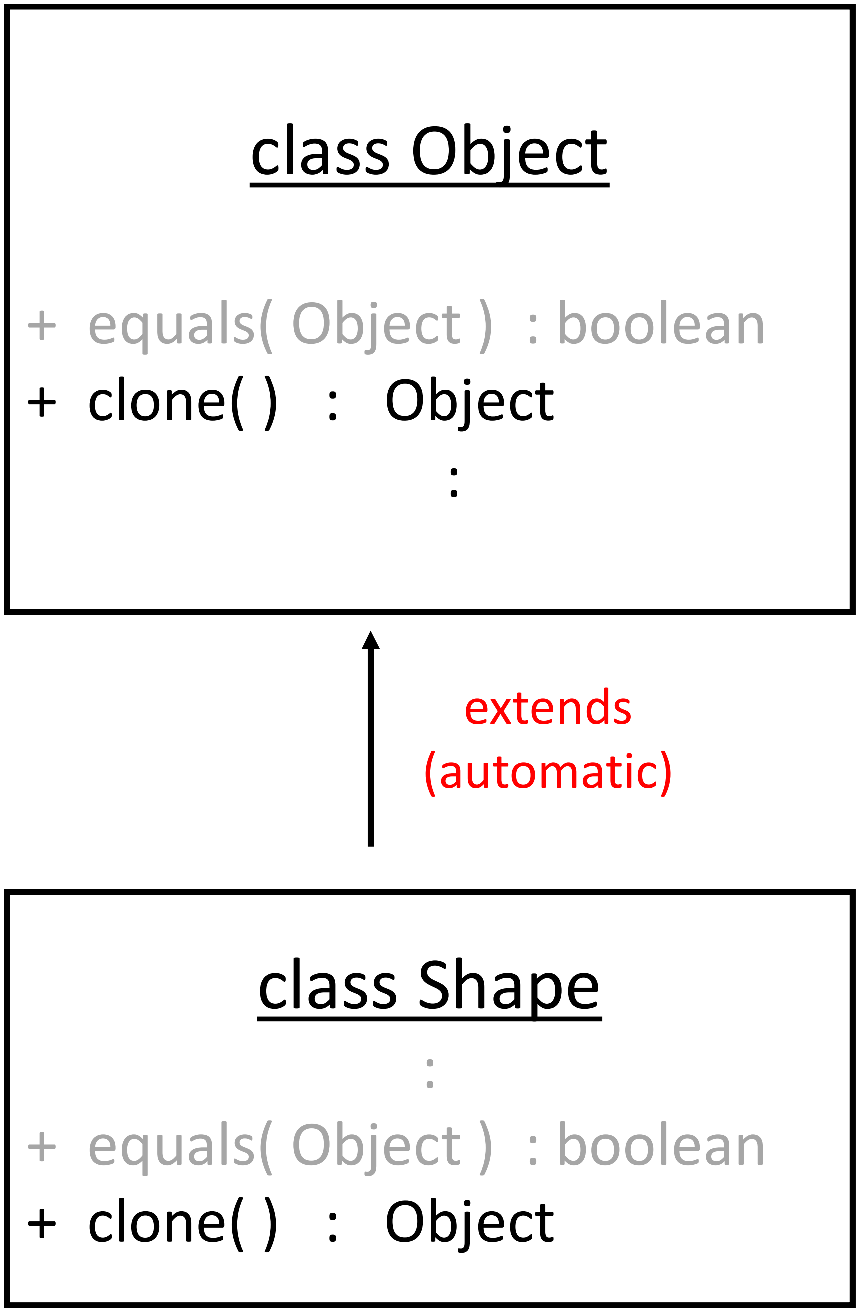
+ clone( ) : Object

+ hashCode( ) : int

+ toString( ) : String

:

makes a new object



class Object

+ equals(Object) : boolean
+ clone( ) : Object
:

extends
(automatic)

class Shape
:
+ equals(Object) : boolean
+ clone( ) : Object

Object.clone()
makes a new object.

When we make our own class, we *don't have to* override the Object clone method, or other Object class methods.

But if we do, then we should be consistent (next slide).

Object.clone() recommendation

Q: x.clone() == x should be true or false ?

A: false

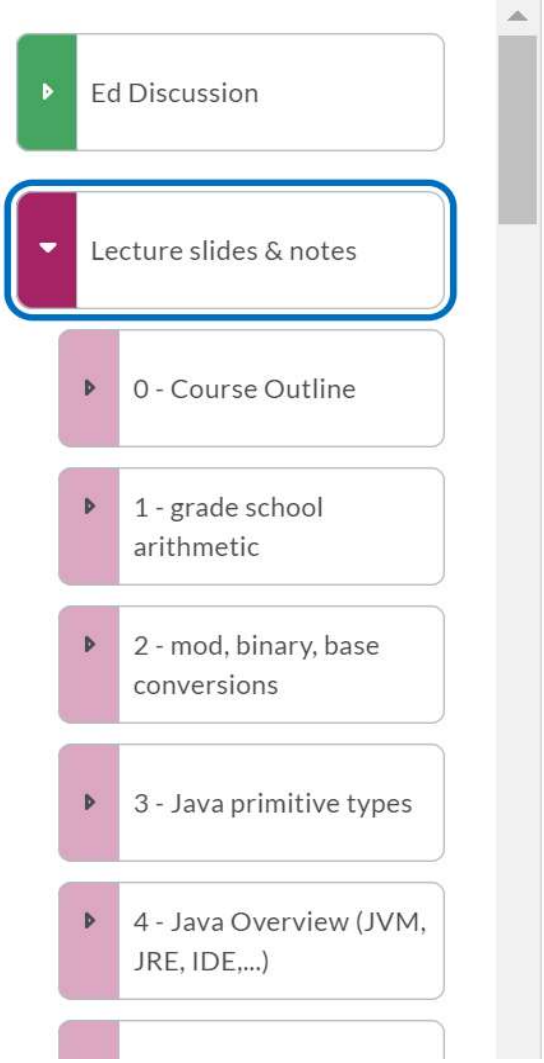
Q: x.equals(x.clone()) should be true or false ?

A: true

equals() needs to be defined to ensure this

[ASIDE: there are some subtleties with when a class can have a clone() method and how to use it. We will avoid these subtleties so that we are not bogged down in details.]

Coming up...



Ed Discussion

Lecture slides & notes

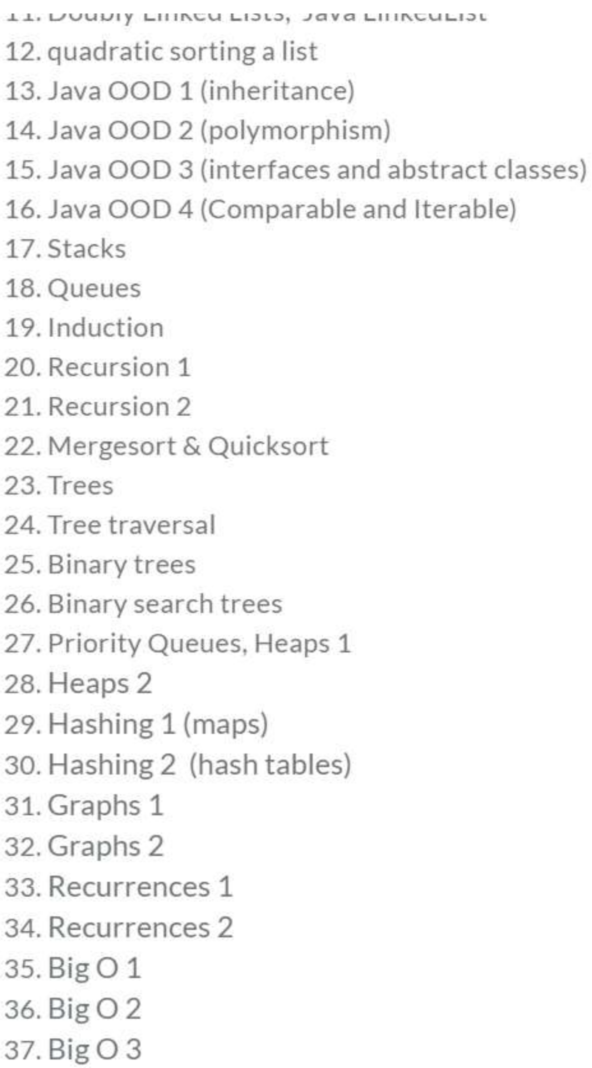
0 - Course Outline

1 - grade school arithmetic

2 - mod, binary, base conversions

3 - Java primitive types

4 - Java Overview (JVM, JRE, IDE,...)



11. Doubly Linked Lists, Java LinkedList
12. quadratic sorting a list
13. Java OOD 1 (inheritance)
14. Java OOD 2 (polymorphism)
15. Java OOD 3 (interfaces and abstract classes)
16. Java OOD 4 (Comparable and Iterable)
17. Stacks
18. Queues
19. Induction
20. Recursion 1
21. Recursion 2
22. Mergesort & Quicksort
23. Trees
24. Tree traversal
25. Binary trees
26. Binary search trees
27. Priority Queues, Heaps 1
28. Heaps 2
29. Hashing 1 (maps)
30. Hashing 2 (hash tables)
31. Graphs 1
32. Graphs 2
33. Recurrences 1
34. Recurrences 2
35. Big O 1
36. Big O 2
37. Big O 3