COMP 250

Lecture 30

inheritance

overriding vs overloading

Nov. 17, 2017

$$\left.\begin{array}{l}\text{All dogs are animals.}\\[2em] \text{All beagles are dogs.}\end{array}\right\}\begin{array}{l}\text{relationships}\\ \text{between}\\ \text{classes}\end{array}$$

All dogs are animals.

All beagles are dogs.

$\Big\}$ relationships between classes

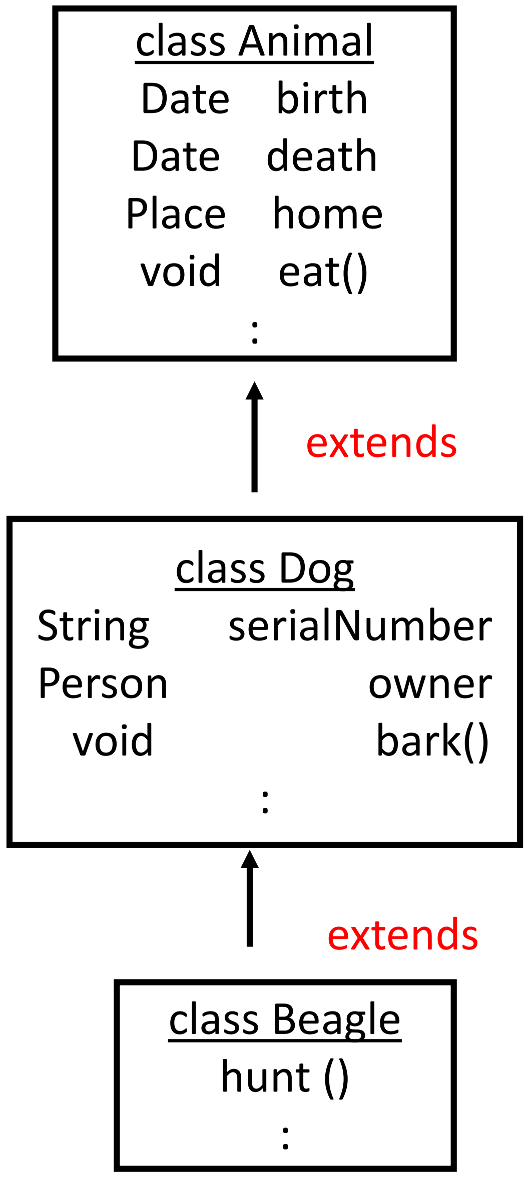
Animals are born (and have a birthdate).

Dogs bark.

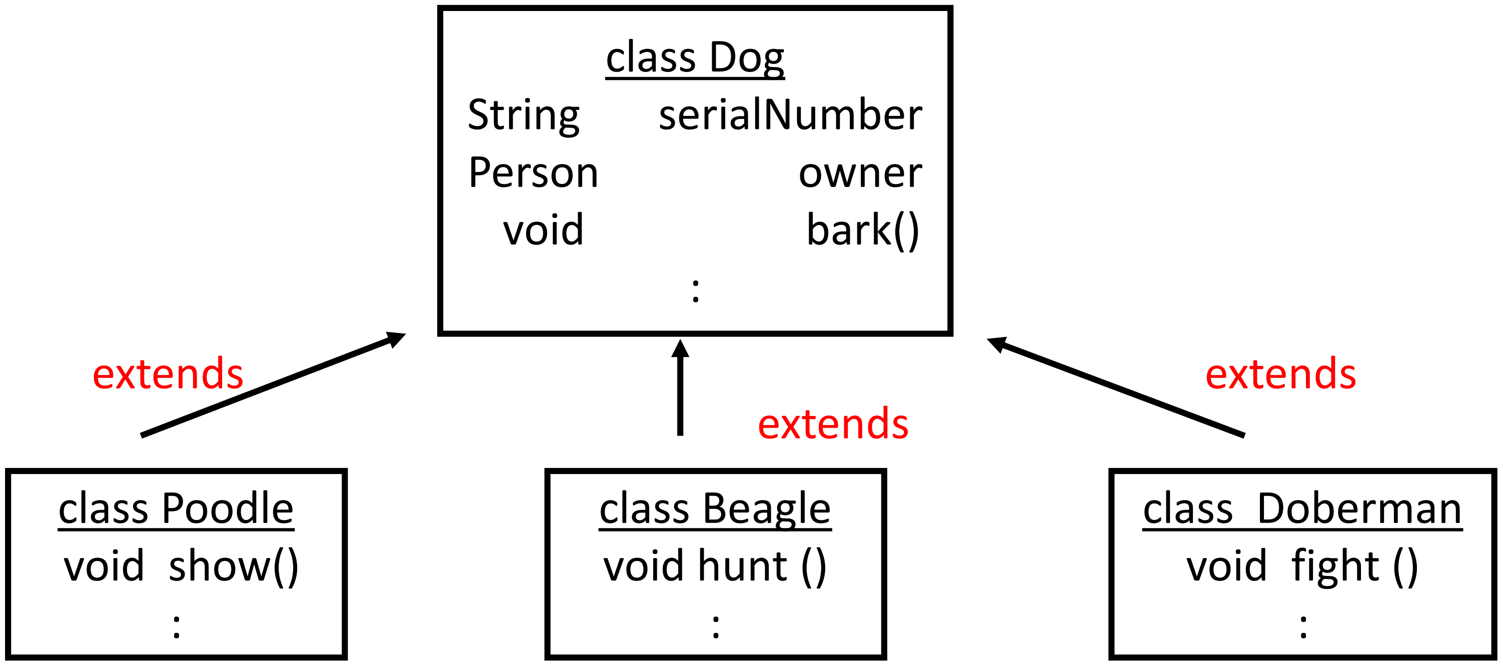
Beagles chase rabbits.

$\Big\}$ class definitions

Inheritance



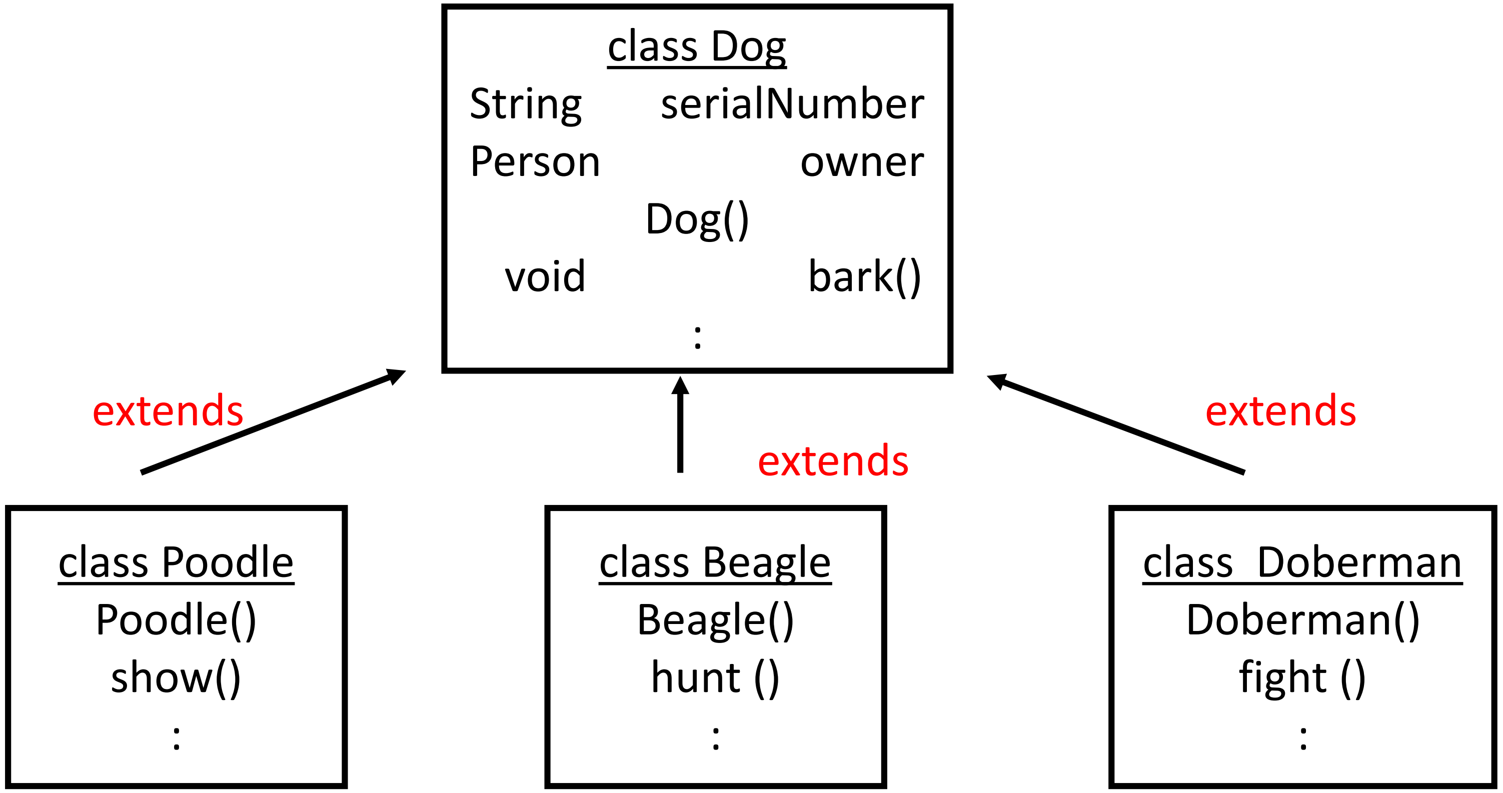
Inheritance



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

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

Constructors are not inherited.



Each object belongs to a unique class.

Constructor chaining

```
class Animal {
    Place  home;

    Animal( ) { ... }

    Animal( Place home) {
        this.home  =  home;
    }
}

class  Dog  extends  Animal {
     String owner;

     Dog( ) {  }    //    This constructor   automatically creates
                    //    fields  that are inherited from the superclass




}
```

Constructor chaining (a few details...)

```
class Animal {
    Place  home;

    Animal() { ... }

    Animal( Place home) {
        this.home  =  home;
    }
}

class  Dog  extends  Animal {
     String owner;

     Dog() {  }    //   This constructor   automatically calls super() which creates
                   //   fields  that are inherited from the superclass

     Dog(Place home,  String  owner) {
        super(home);                          //   Here we need to explicitly write it.
        this.owner  = owner;
     }
          :
}
```

Sometimes we have two versions of a method:

(method) overloading

vs.

(method) overriding

Today we will see some examples.

The reasons why we do this will hopefully become more clear over the next few lectures.

Example of overloading
LinkedList<E>

```
void  add( E  e)
void  add( int  index,  E    e )

E       remove( int  index)
E       remove( )                          //   removes head
```

Overloading

- same method name, but different parameter types

  (i.e. different method "signature"
  note: "signature" does not include the return type)

- *within* a class, or *between* a class and its superclass

  Example on previous slide was *within* a class

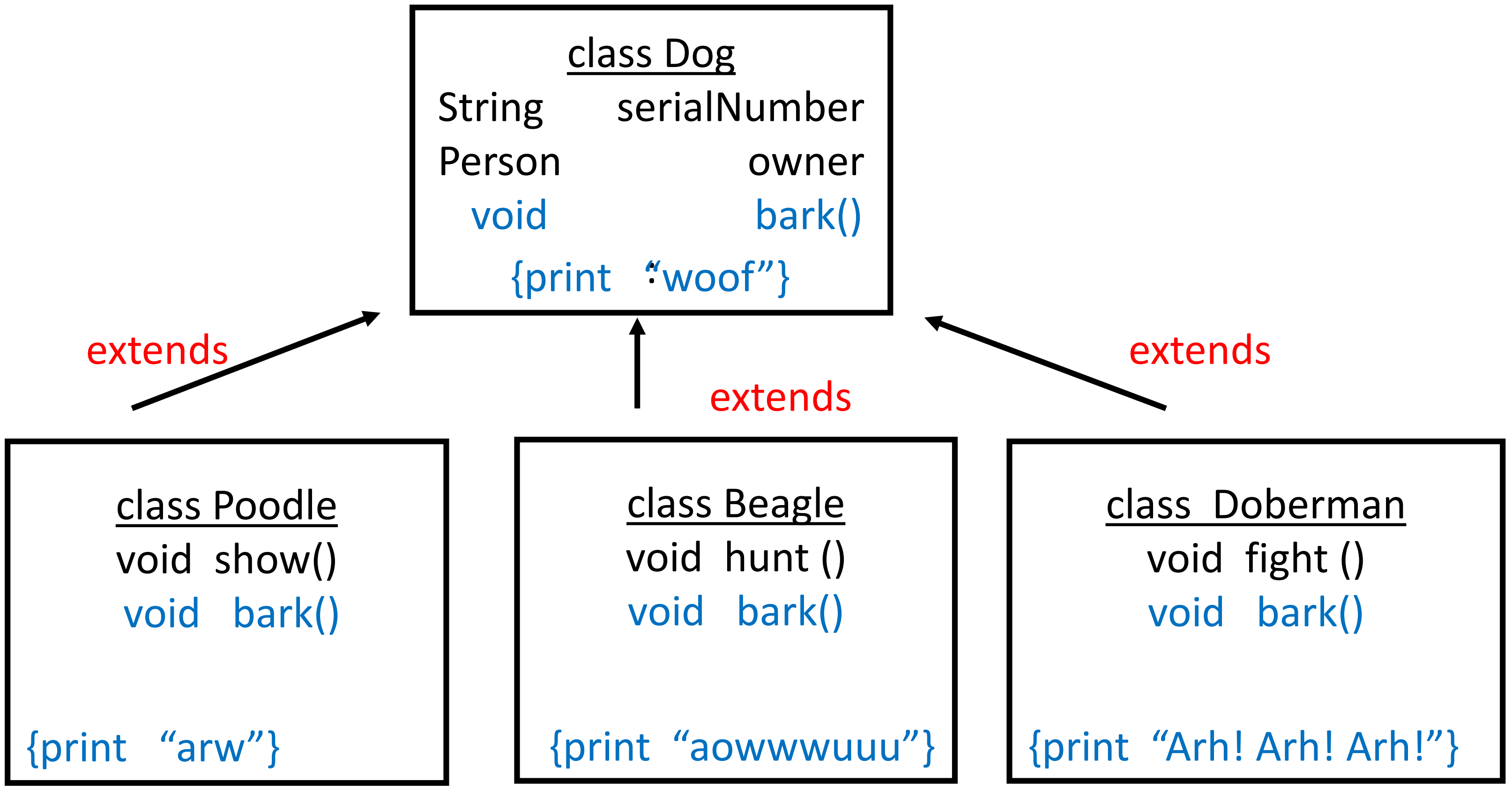
Overriding

- subclass method *overrides* a superclass method

- same method signatures

  (i.e. same method name and parameter types)

Overriding e.g. bark()

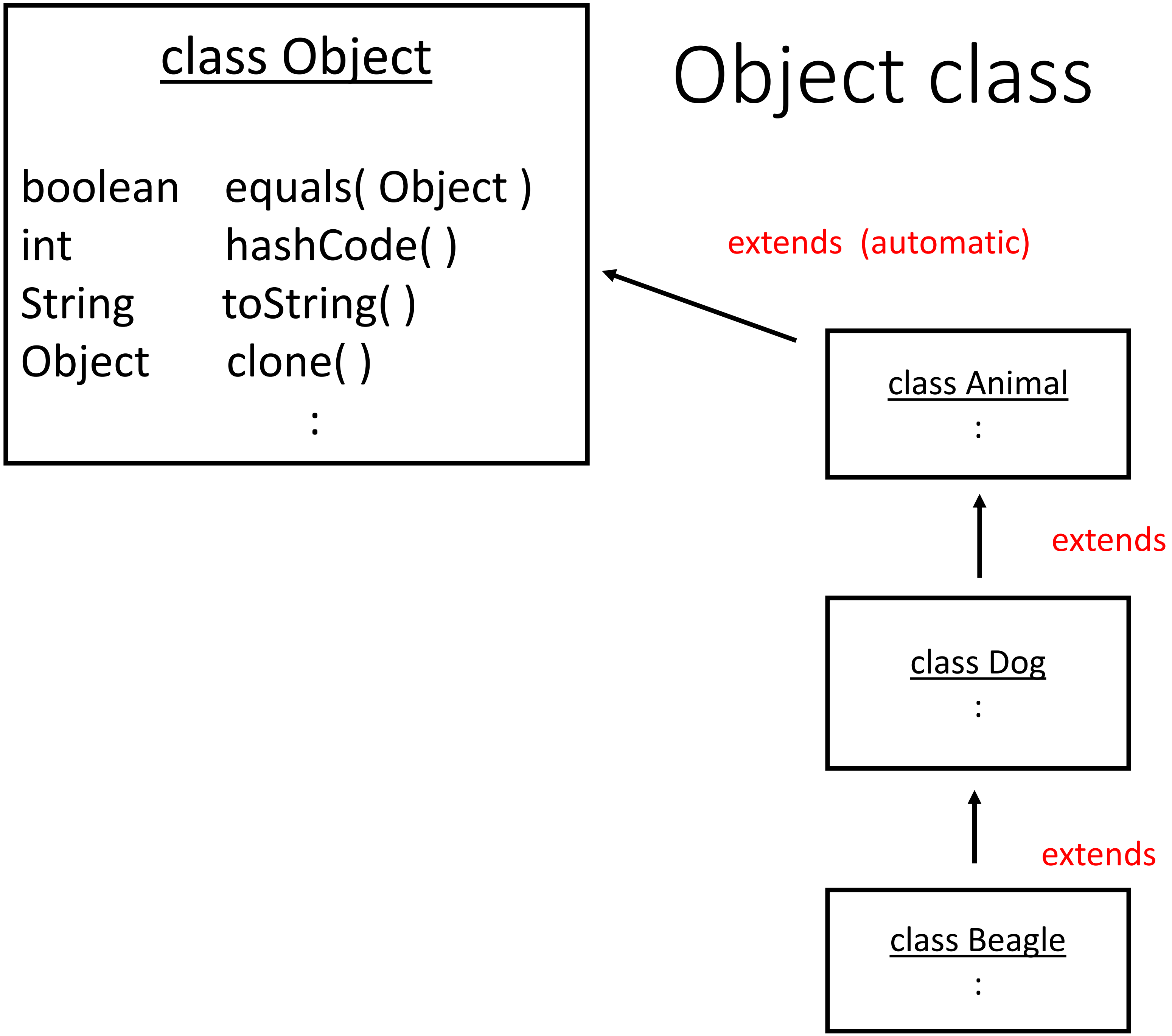


https://www.youtube.com/watch?v=_wqK15EtCMo

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

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

Object class



class Object

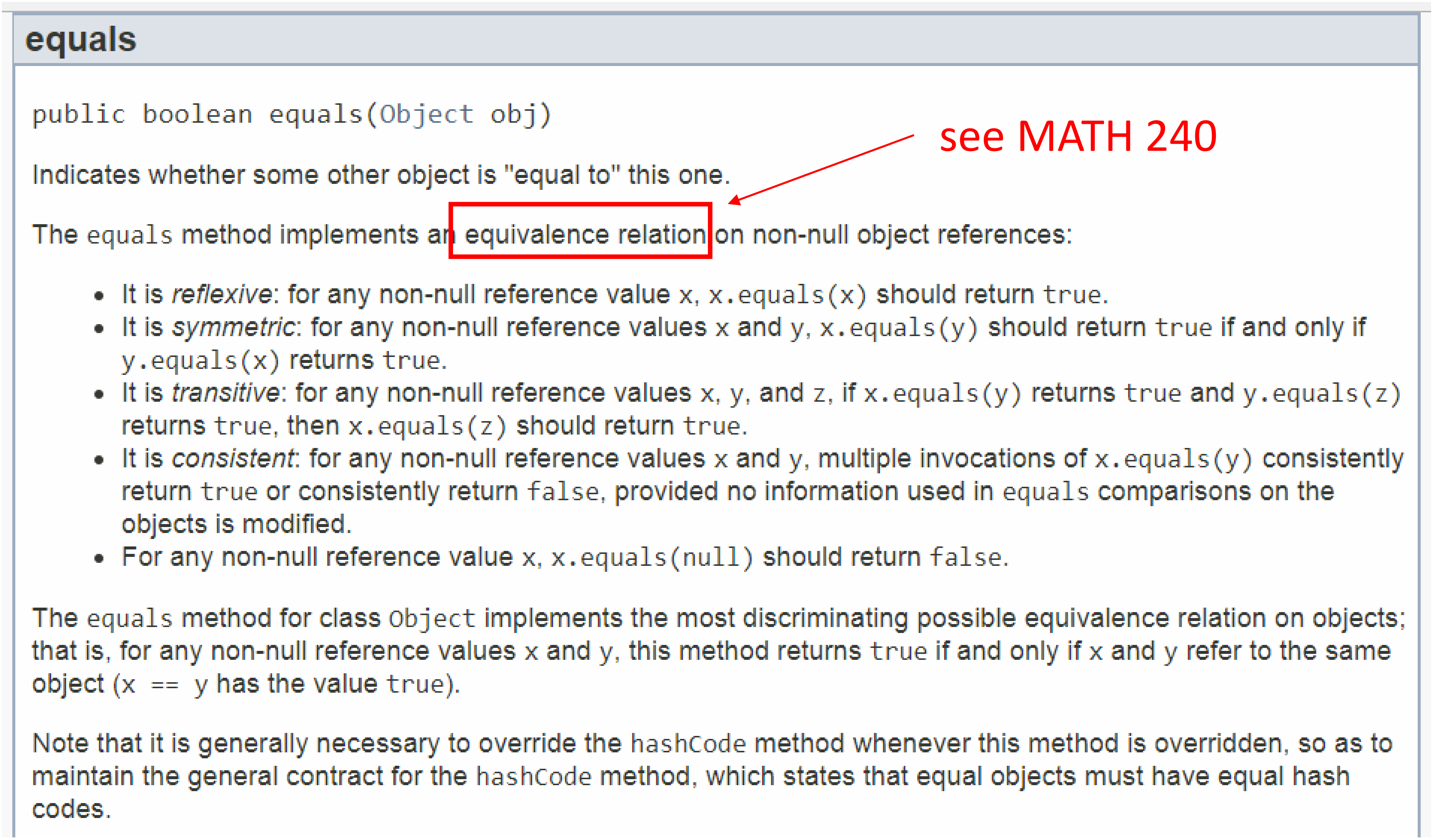
boolean	equals(Object)
int	hashCode( )
String	toString( )
Object	clone( )

:

Object.equals(Object)

Object   obj1,   obj2

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

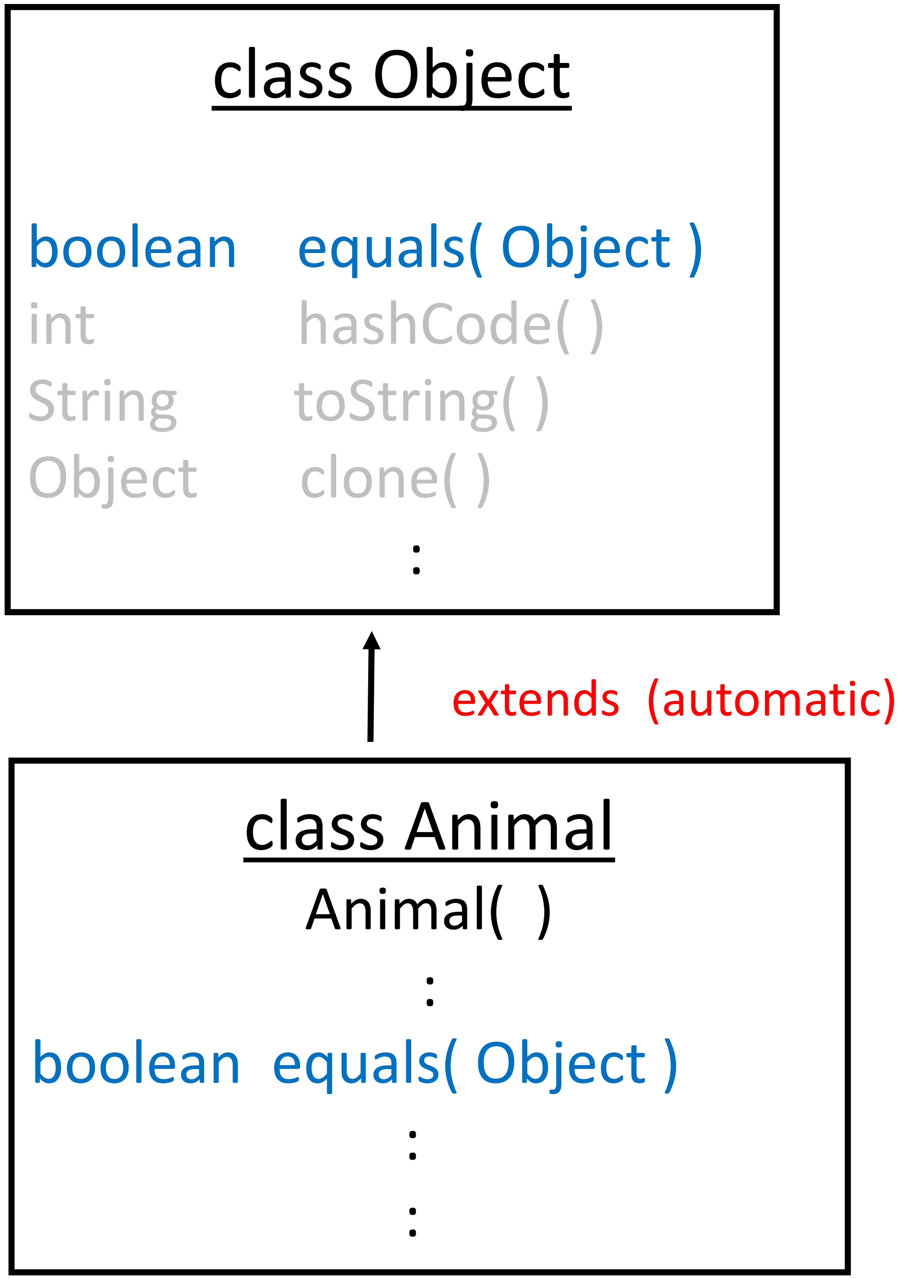
x.equals(x)   should always return true

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.

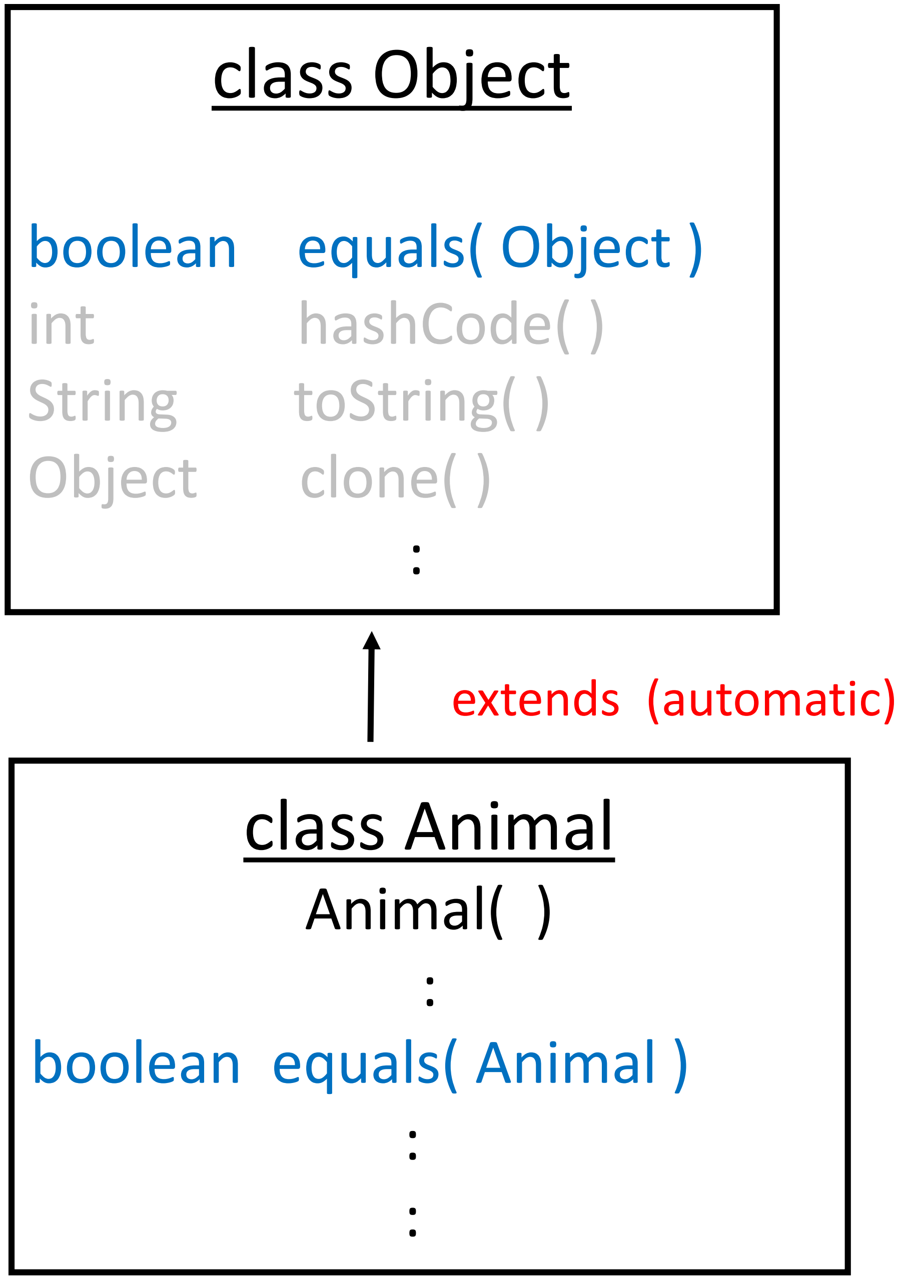
The above rules should hold for non-null references.



Object.equals(Object)

Animal.equals(Object)

This is overriding.



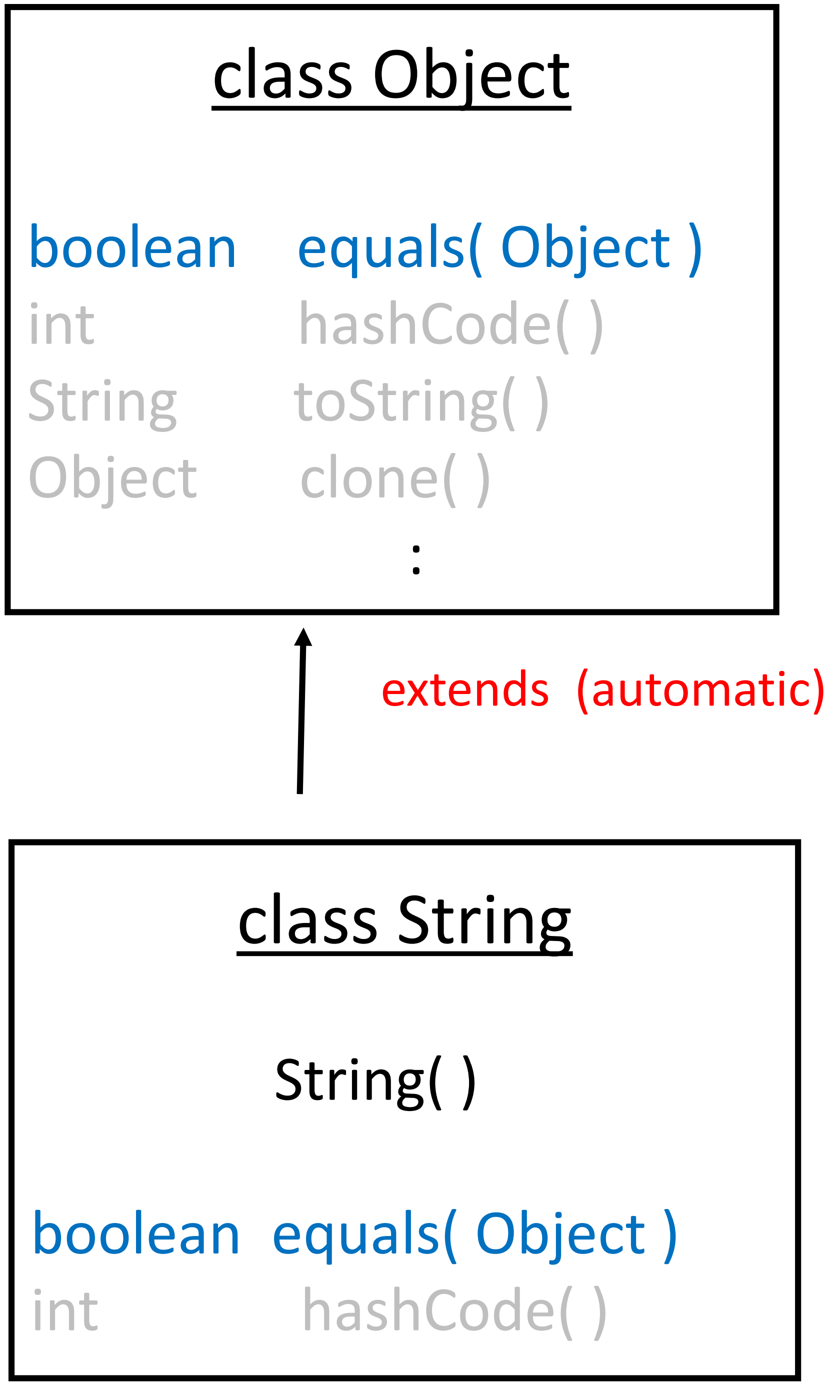
Object.equals(Object)

Animal.equals(Animal)

This is overloading.

overloading
vs.
overriding

I will say a bit more about when we use one versus the other over the next few lectures.



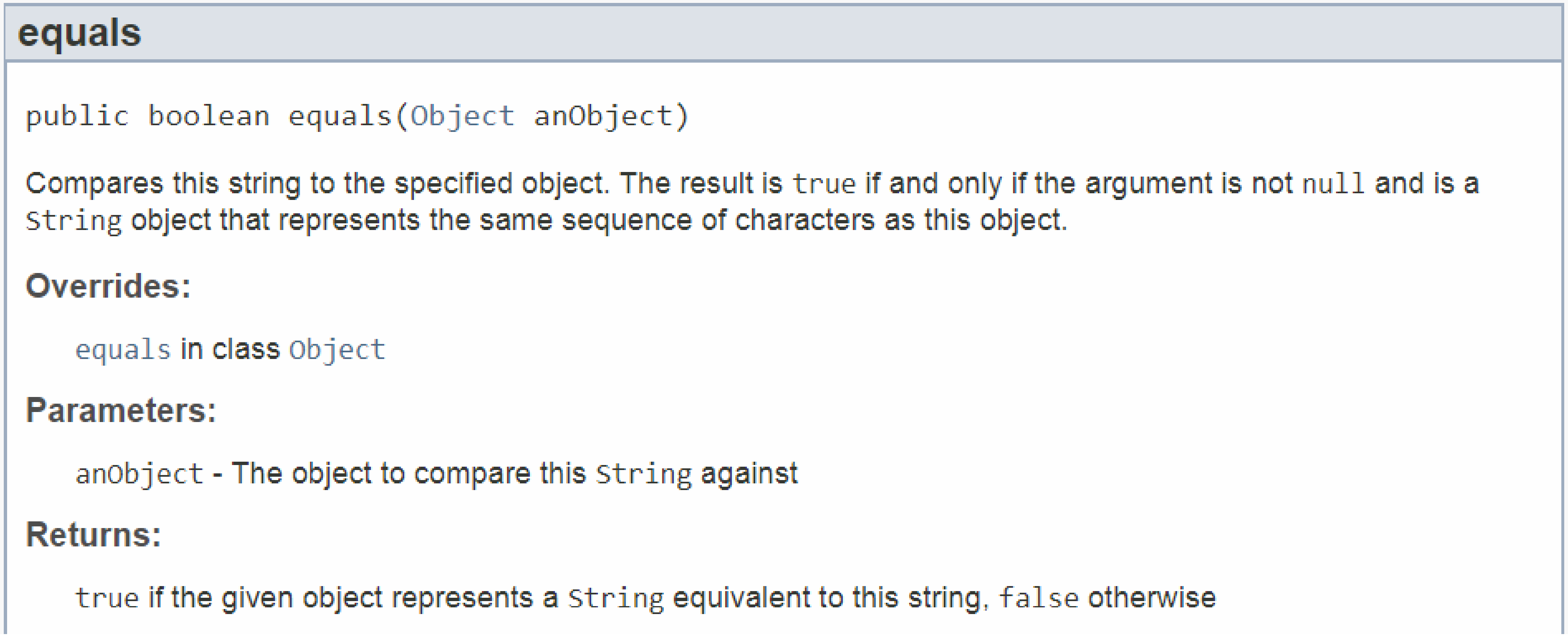
Object.equals(Object)

String.equals(Object)

This is overriding.

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:

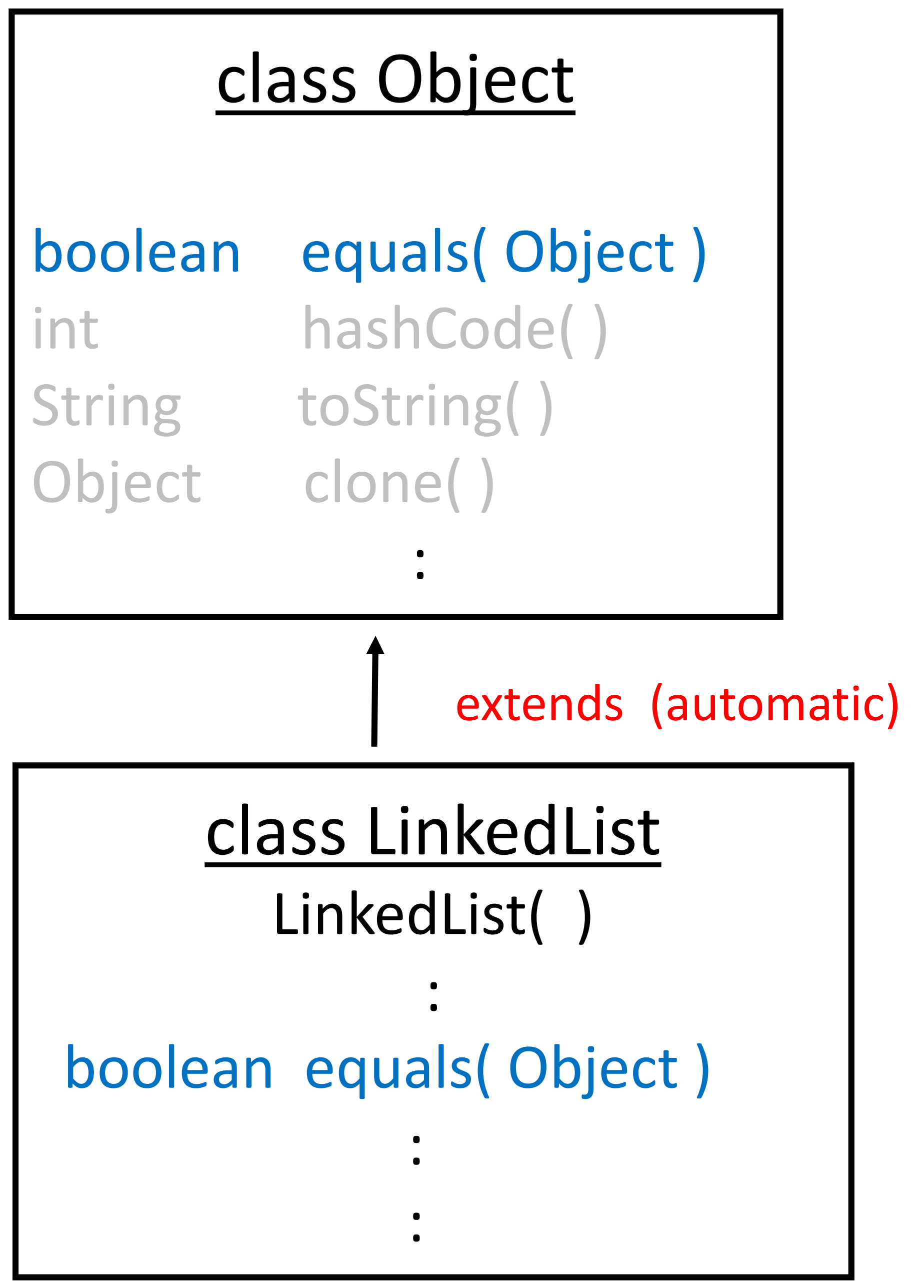
`true` if the given object represents a `String` equivalent to this string, `false` otherwise

You should Compare strings using String.equals(Object) rather than "==" to avoid nasty surprises.

```
String s1 = "sur";
String s2 = "surprise";

System.out.println(("sur" + "prise") == "surprise");              // true
System.out.println("sur" == s1);                                  // true
System.out.println(("surprise" == "surprise"));                   // true
System.out.println("surprise" == new String("surprise"));         // false
System.out.println((s1 + "prise") == "surprise");                 // false
System.out.println((s1 + "prise") == s2);                         // false

System.out.println((s1 + "prise").equals("surprise"));            // true
System.out.println((s1 + "prise").equals(s2));                    // true
System.out.println( s2.equals(s1 + "prise"));                     // true
```



Object.equals(Object)

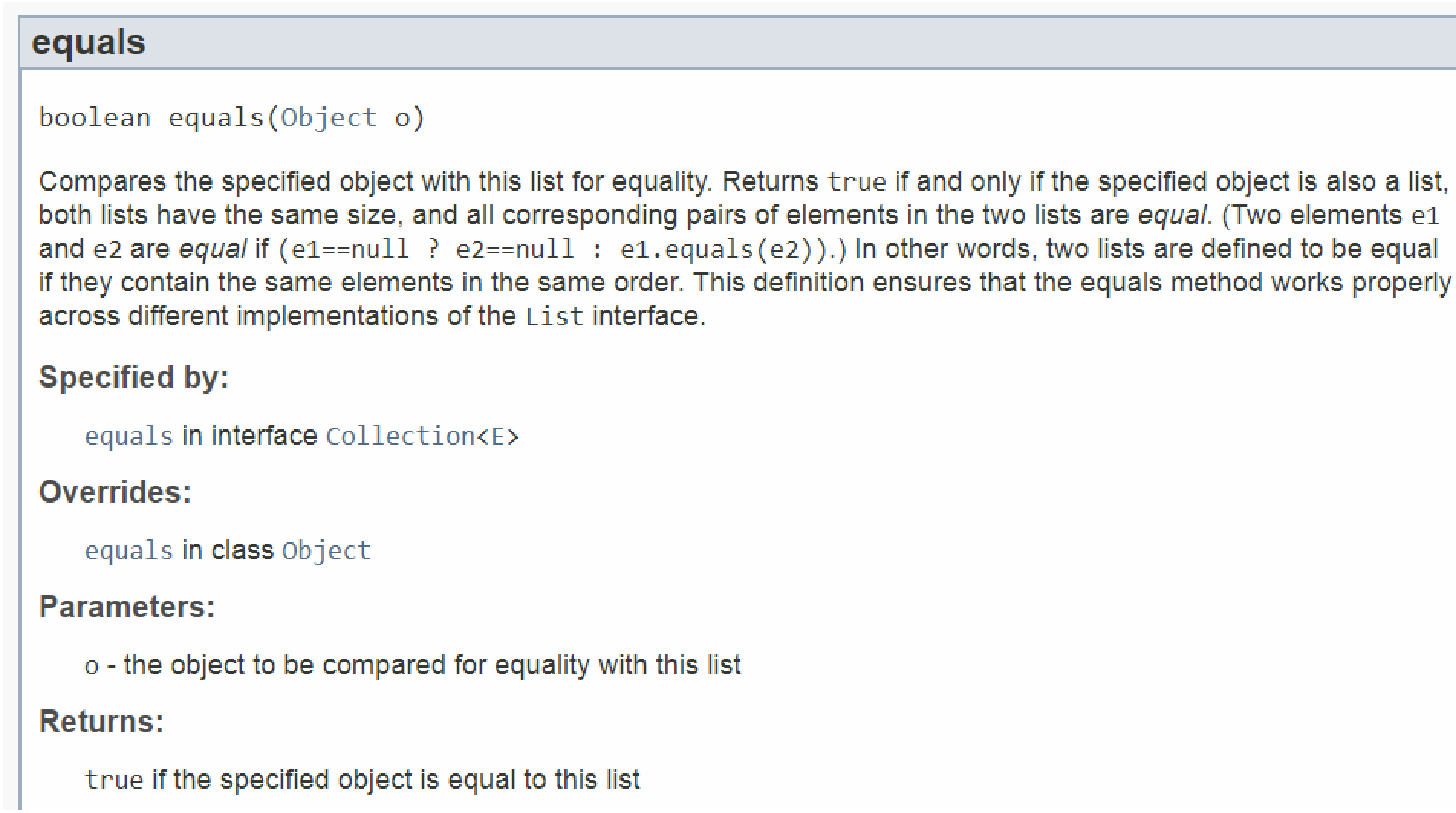
LinkedList.equals(Object)

This is overriding.

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

LinkedList.equals(Object)

List interface: next lecture



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:

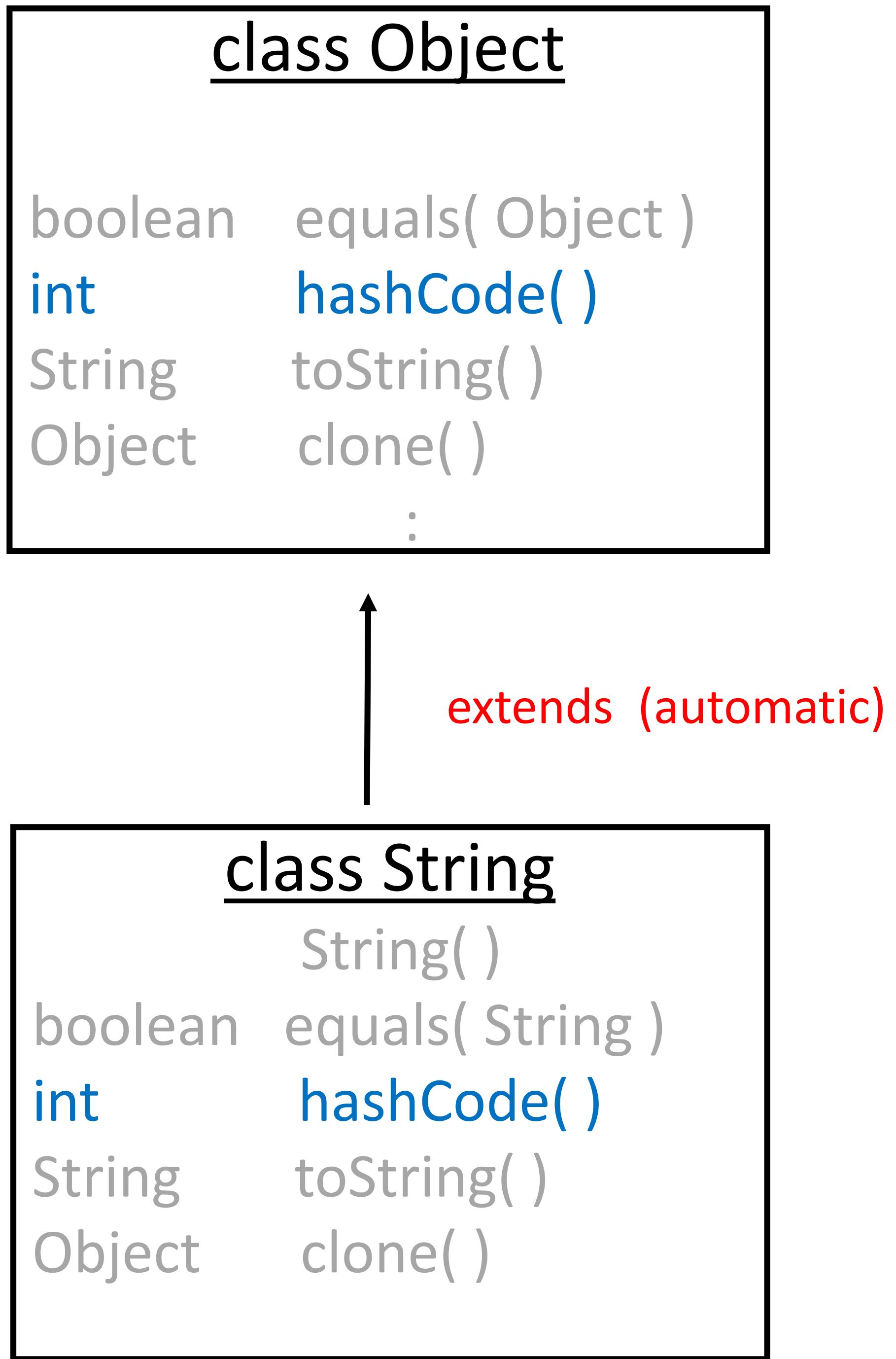
`o` - the object to be compared for equality with this list

Returns:

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

class Object

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



Object.hashCode()

← Returns a 32 bit integer

String.hashCode()

This is overriding.

SLIDE ADDED Nov. 20
(I will discuss this next lecture too)

Java API for Object.hashCode() recommends:

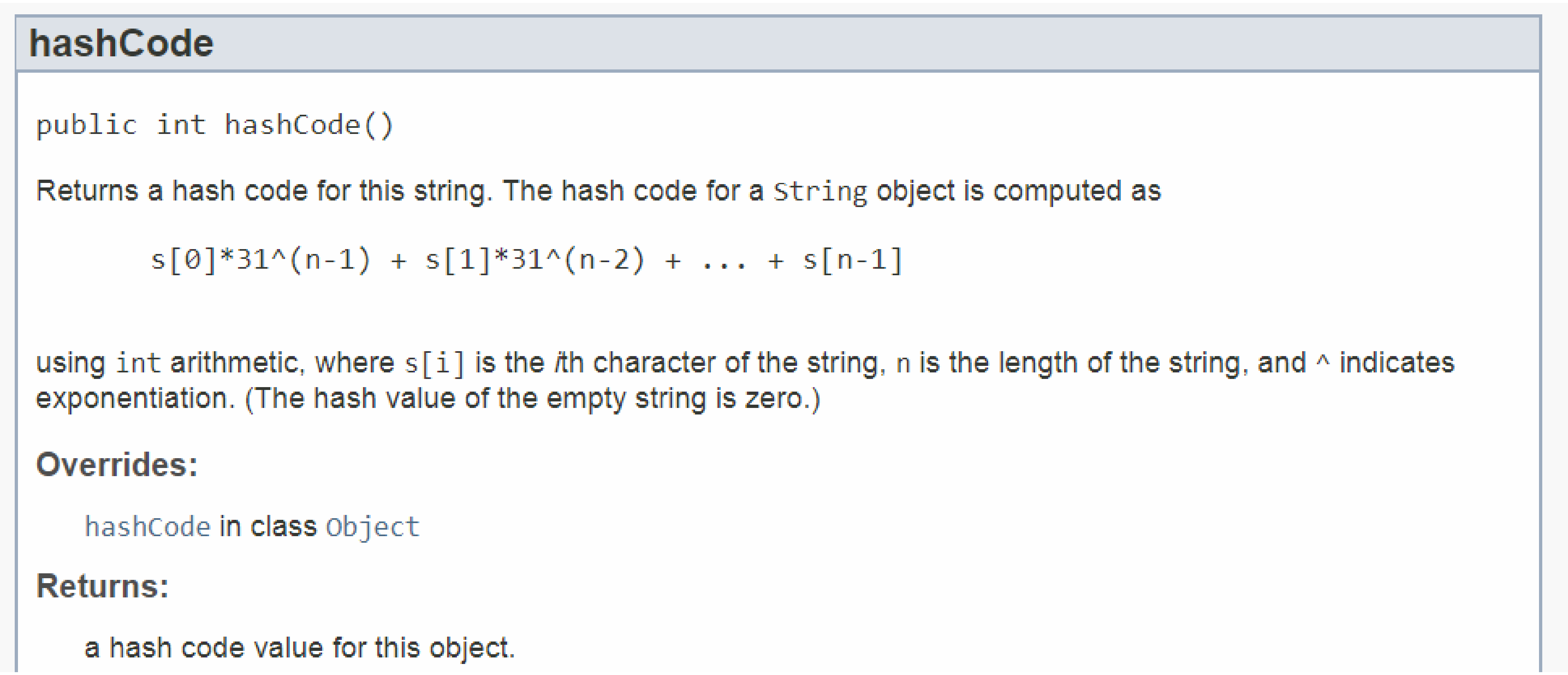
If   o1.equals(o2) is true then

o1.hashCode() == o2.hashCode() should be true.

The converse need not hold.   It can easily happen that two objects have the same hashCode but the objects are not considered equal.

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

String.hashCode()



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:

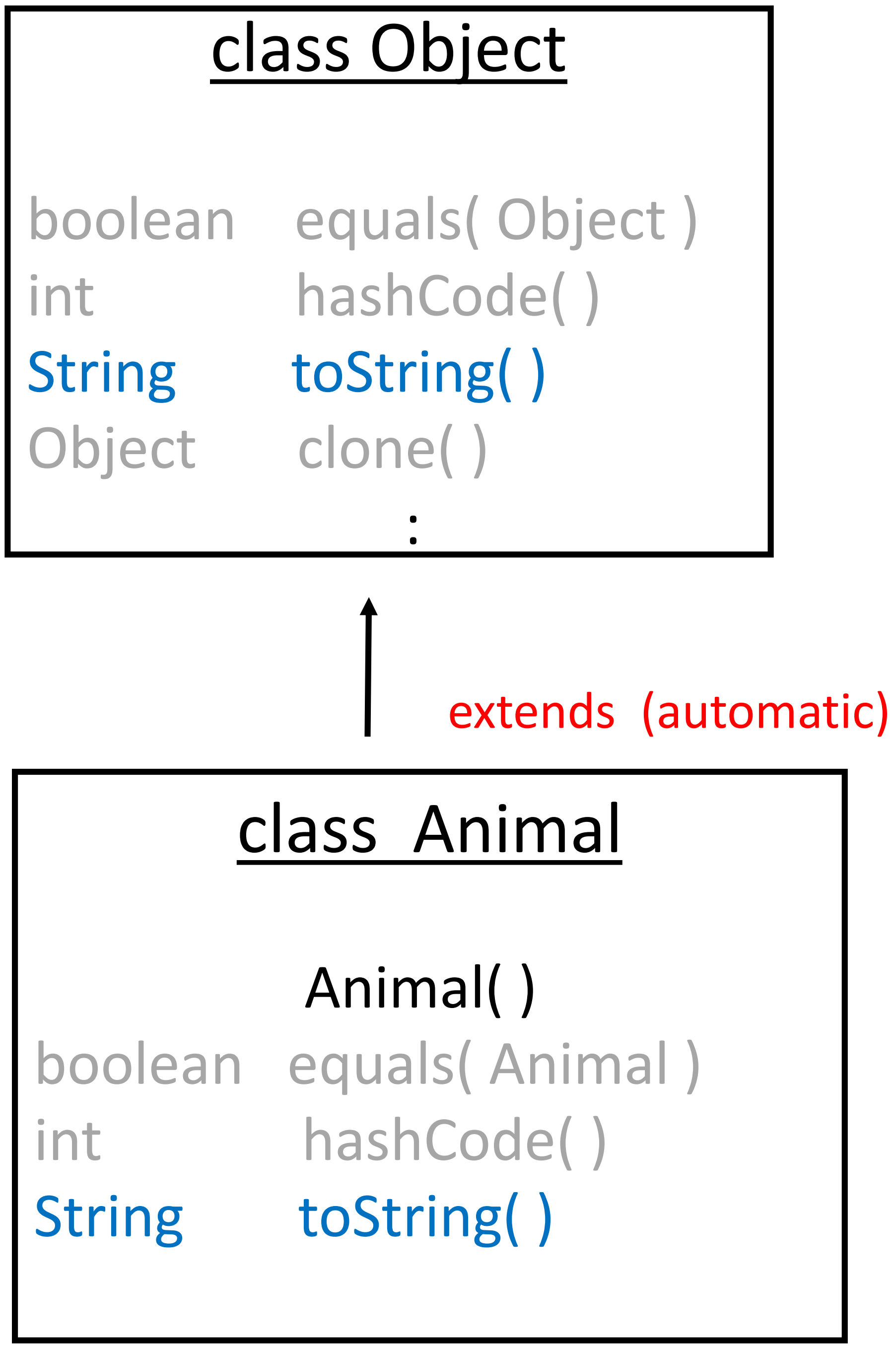
a hash code value for this object.

For fun,   check out hashcode() method for other classes e.g. LinkedList.

class Object

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

:



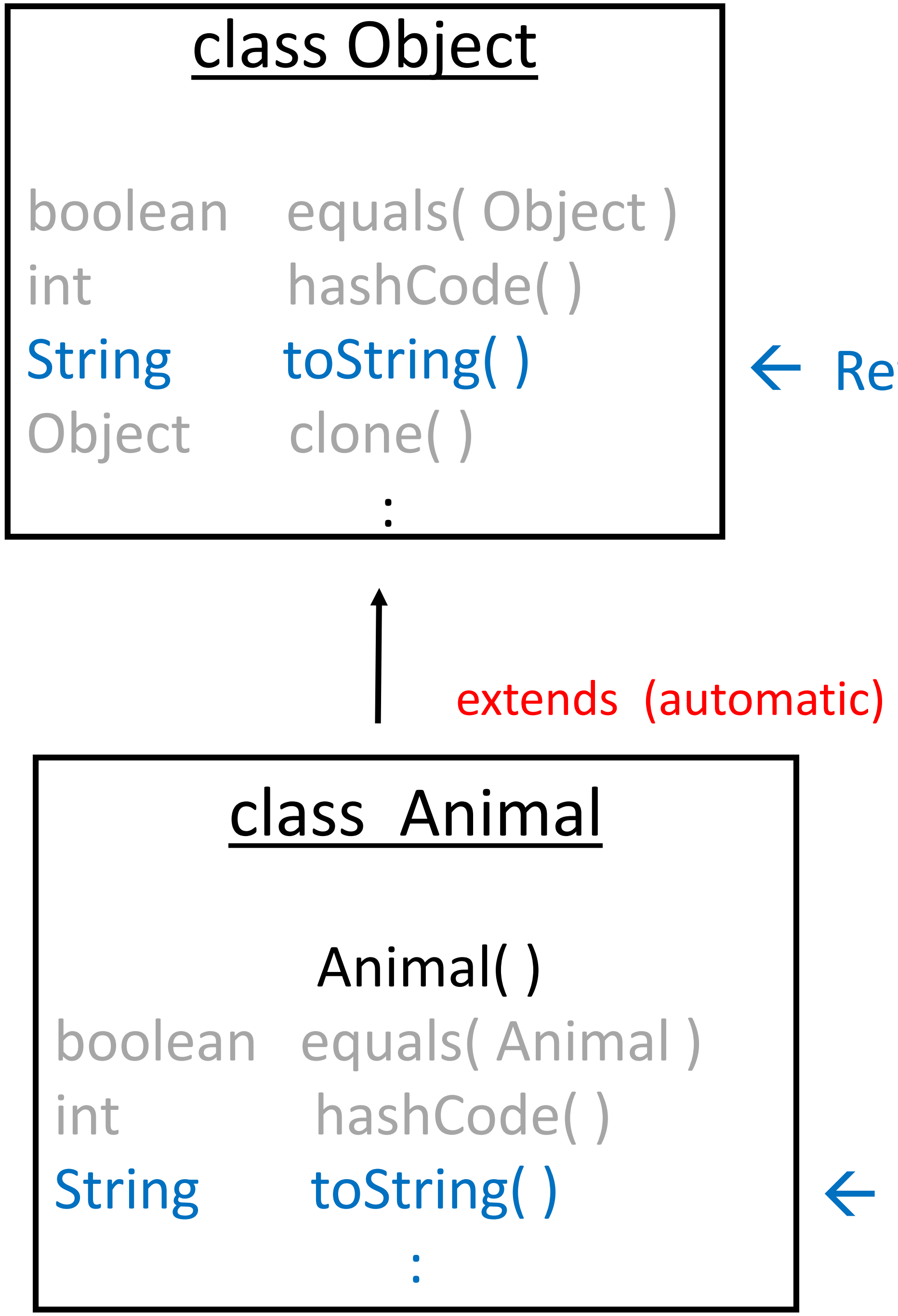
Object.toString()

← Returns ?

Animal.toString()

This is overriding.

← Returns ?



Object.toString()

← Returns classname + "@" + hashCode()

Animal.toString()

This is overriding.

← Returns .... however you define it

Object.toString()

returns classname + “@” + hashCode()

In order to explain this, I need to take a detour.

I have also added the following slides to lecture 2 (binary numbers).   That is really where the following material belongs.

As you know from Assignment 1, we can write any positive integer m uniquely as a sum of powers of any number called the *base* (or *radix).*

$$m = \sum_{i=0}^{N-1} a_i \, (base)^i$$

The coefficients a_i are in {0, 1, ….., $base$ – 1}

We write $(a_{N-1} \; a_{N-2} \; a_{N-3} \; \ldots \; a_2 \; a_1 \; a_0 \;)_{base}$

Humans usually use base 10. Computers use base 2.

e.g. Hexadecimal (base 16)

$$m = \sum_{i=0}^{N-1} a_i \, (16)^i$$

The coefficients a_i are in {0, 1, ….., 10, 11, … 15}

Instead we use a_i in {0, 1, ….., 8, 9, a, b, c, d, e, f}

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Common use of hexadecimal: representing long bit strings

Example: 0010 1111 1010 0011

2 f a 3

Example 2: 10 1100

We write 2c (10 1100), not b0 (1011 00).

Object.toString()

Returns classname + "@" + hashCode()

32 bit integer
(8 hexadecimal digits)
Address of object

In Eclipse, we get the package name also.

Object.toString()

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

What does this print?

Object.toString()

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

What does this print?

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

$\uparrow$

32 bit integer represented in hexadecimal.
You'll get a different number if you run it again.

Object.toString()

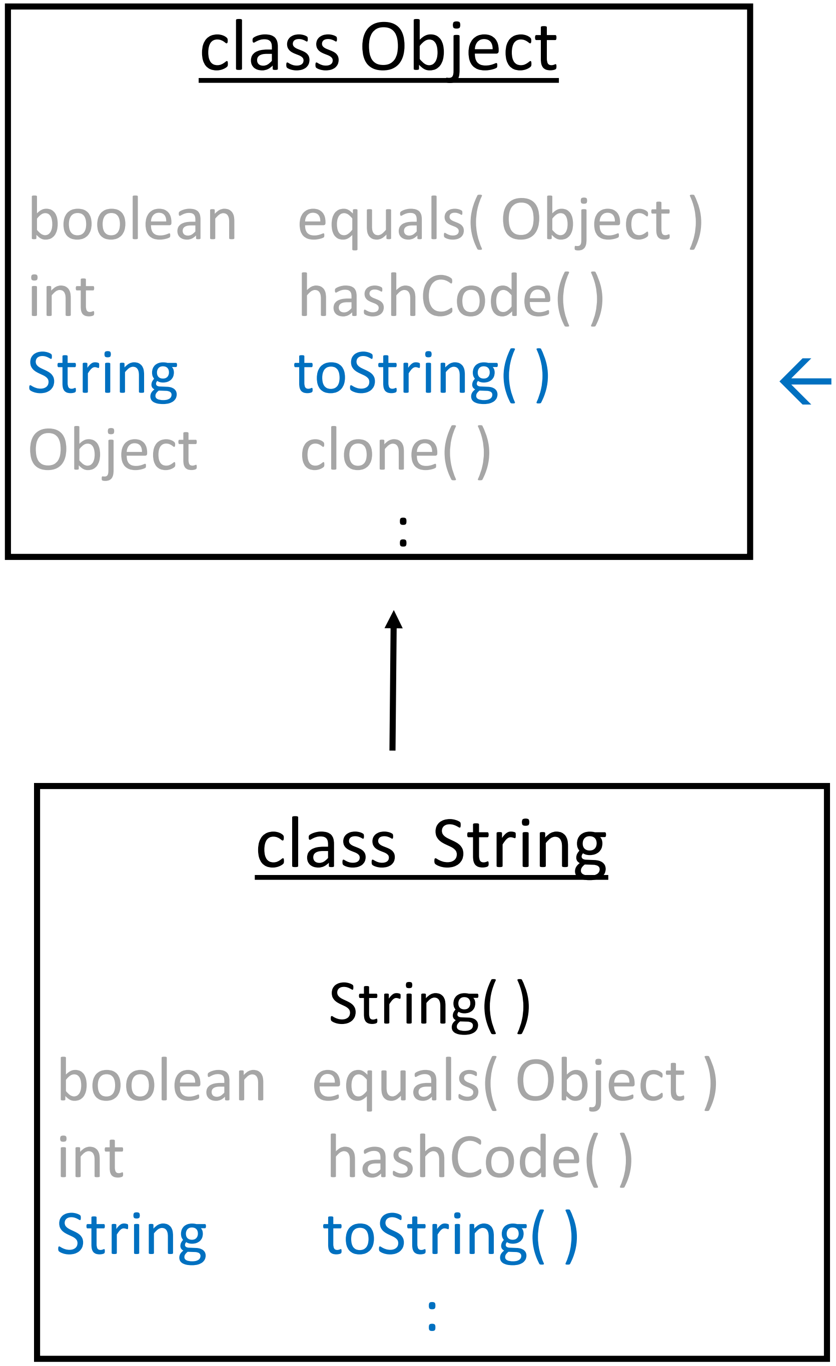
```
Object o = new Object();
System.out.println( o );
```

What does this print?

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

package + class name

32 bit integer represented in hexadecimal.
You'll get a different number if you run it again.



class Object

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

class String

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

Object.toString()

← Returns classname + “@” + hashCode()

String.toString()

This is overriding.

← Returns itself!