COMP 250
Lecture 30

inheritance

overriding vs overloading

Nov. 17, 2017
All dogs are animals.

All beagles are dogs.

relationships between classes
All dogs are animals.

All beagles are dogs.

Animals are born (and have a birthdate).

Dogs bark.

Beagles chase rabbits.
Inheritance

class Animal
  Date    birth
  Date    death
  Place   home
  void    eat()
:

extends

class Dog
  String  serialNumber
  Person  owner
  void    bark()
:

extends

class Beagle
  hunt ()
:


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.
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

}
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()`

```
class Dog
    String serialNumber
    Person owner
    void bark()
        {print "woof"}

extends
```

```
class Beagle
    void hunt()
    void bark()
        {print "aowwwuuu"}

extends
class Poodle
    void show()
    void bark()
        {print "arw"}

extends
class Doberman
    void fight()
    void bark()
        {print "Arh! Arh! Arh!"}
```

https://www.youtube.com/watch?v=wgK15EtCMo

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

https://www.youtube.com/watch?v=s5YGyt57Dw
class Object

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

extends (automatic)

class Animal

extends

class Dog

extends

class Beagle
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
equals

public boolean equals(Object obj)

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

The equals method implements an \textit{equivalence relation} on non-null object references:

- \textit{It is reflexive}: for any non-null reference value \( x \), \( x.equals(x) \) should return true.
- \textit{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.
- \textit{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.
- \textit{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 \texttt{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 \texttt{hashCode} method whenever this method is overridden, so as to maintain the general contract for the \texttt{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.
class Object

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

extends (automatic)

class Animal

Animal( )

boolean equals( Object )

Animal.equals( Object)
This is overriding.
class Object

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

extends (automatic)

class Animal

Animal()

boolean equals(Animal )

This is overloading.

Object.equals( Object )

Animal.equals( Animal )
I will say a bit more about when we use one versus the other over the next few lectures.
class Object

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

extends (automatic)

class String

String( )

boolean equals( Object )
int hashCode( )

Object.equals( Object )

This is overriding.

String.equals( Object )

This is overriding.
String.equals( Object )

<table>
<thead>
<tr>
<th>equals</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean equals(Object anObject)</td>
</tr>
</tbody>
</table>

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.

```java
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
```
class Object

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

extends (automatic)

class LinkedList

LinkedList():

boolean equals(Object)

This is overriding.

Object.equals(Object)

LinkedList.equals(Object)

This is overriding.
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( )

:
class Object

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

extends (automatic)

class String

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

Object.hashCode() ← Returns a 32 bit integer

String.hashCode() This is overriding.
Java API for Object.hashCode() recommends:

If \(\text{o1.equals(o2)}\) is true then

\(\text{o1.hashCode() == o2.hashCode()}\) 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.
String.hashCode()

```java
public int hashCode()

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

\[ s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \ldots + s[n-1] \]

using int arithmetic, where \( s[i] \) is the \( i \)th character of the string, \( n \) is the length of the string, and \( ^{\wedge} \) 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( )

:
class Object

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

extends (automatic)

class Animal

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

Animal.toString()

← Returns ?

This is overriding.

← Returns ?
class Object

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

extends (automatic)

class Animal

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

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).

\[
m = \sum_{i=0}^{N-1} a_i (\text{base})^i
\]

The coefficients \( a_i \) are in \( \{0, 1, \ldots, \text{base} - 1\} \)

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

e.g. Hexadecimal (base 16)

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

The coefficients \( a_i \) are in \( \{0, 1, \ldots, 10, 11, \ldots, 15\} \)

Instead we use \( a_i \) in \( \{0, 1, \ldots, 8, 9, a, b, c, d, e, f\} \)
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>b</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>c</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>d</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>e</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>f</td>
</tr>
</tbody>
</table>
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

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.


```java
class Object
{
  boolean equals( Object )
  int hashCode()

  String toString()
}

class String
{
  String( )

  boolean equals( Object )
  int hashCode()

  String toString()
}
```

Object.toString()

← Returns `classname + "@" + hashCode()`.

String.toString()

This is overriding.

← Returns itself!