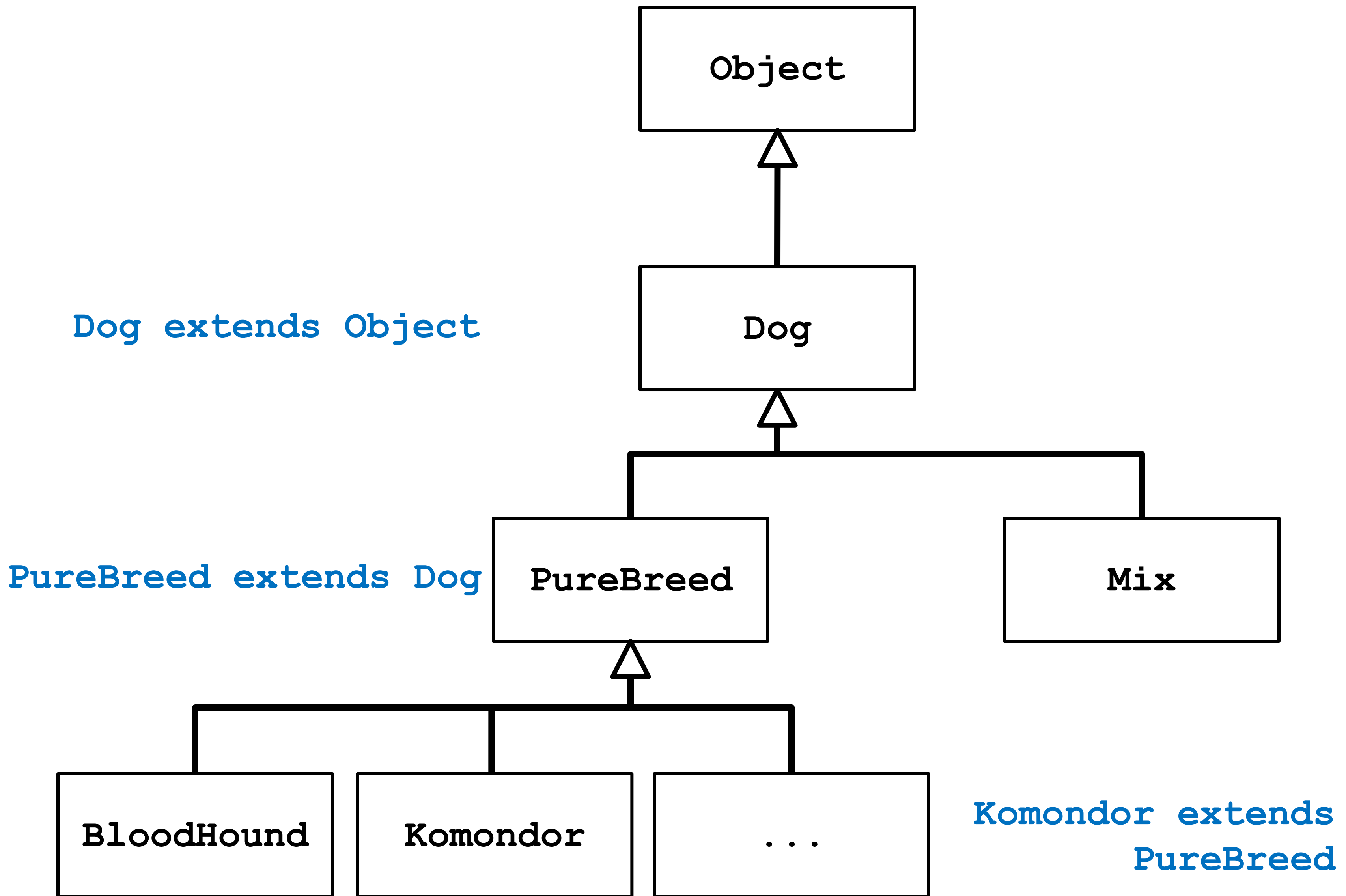
Inheritance (Part 2)

Notes Chapter 6


Object
Dog extends Object
Dog
PureBreed extends Dog
PureBreed
Mix
BloodHound
Komondor
. . .
Komondor extends PureBreed

Implementing Inheritance

- suppose you want to implement an inheritance hierarchy that represents breeds of dogs for the purpose of helping people decide what kind of dog would be appropriate for them
- many possible fields:
  - appearance, size, energy, grooming requirements, amount of exercise needed, protectiveness, compatibility with children, etc.
  - we will assume two fields measured on a 10 point scale
    - size from 1 (small) to 10 (giant)
    - energy from 1 (lazy) to 10 (high energy)

Dog

```
public class Dog extends Object
{
  private int size;
  private int energy;

  // creates an "average" dog
  Dog()
  {  this(5, 5); }

  Dog(int size, int energy)
  {  this.setSize(size);  this.setEnergy(energy);  }
```

```
  public int getSize()
  { return this.size; }

  public int getEnergy()
  { return this.energy; }

  public final void setSize(int size)
  { this.size = size; }

  public final void setEnergy(int energy)
  { this.energy = energy; }
}
```

why final? stay tuned…

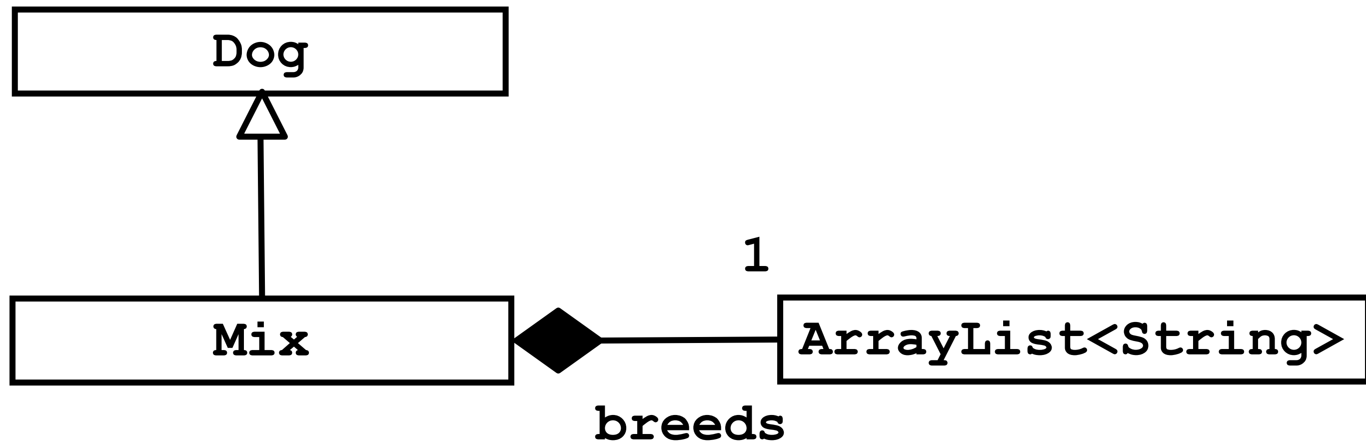
What is a Subclass?

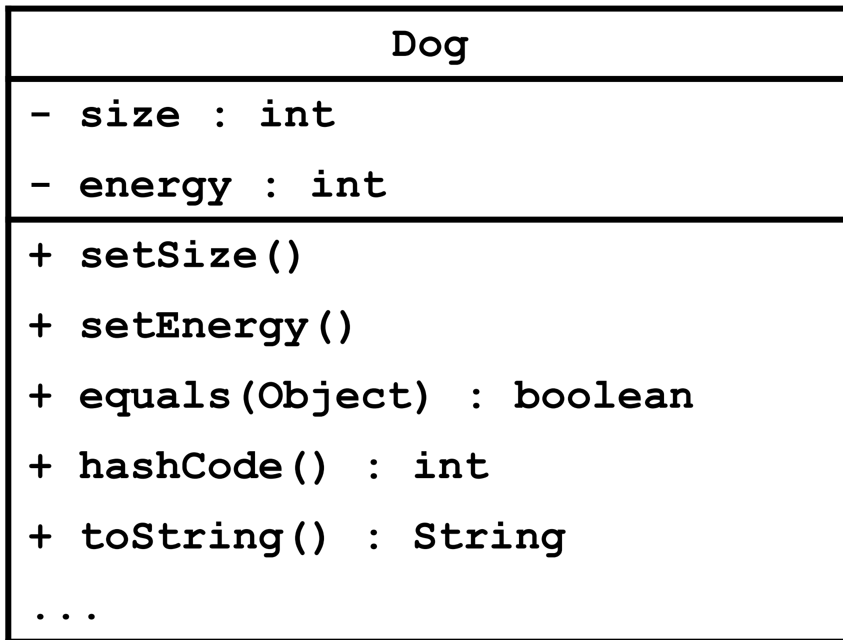
- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
  - the new class has direct access to the **`public`** and **`protected`*** fields and methods without having to re-declare or re-implement them
  - the new class can introduce new fields and methods
  - the new class can re-define (override) its superclass methods

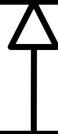
* the notes does not discuss **`protected`** access

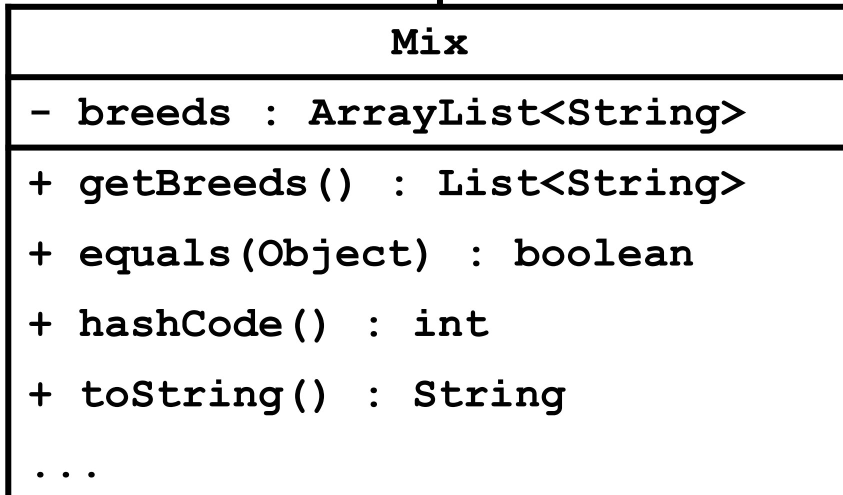
Mix UML Diagram

- a mixed breed dog is a dog whose ancestry is unknown or includes more than one pure breed









- subclass can add new fields
- subclass can add new methods
- subclass can change the implementation of inherited methods

What is a Subclass?

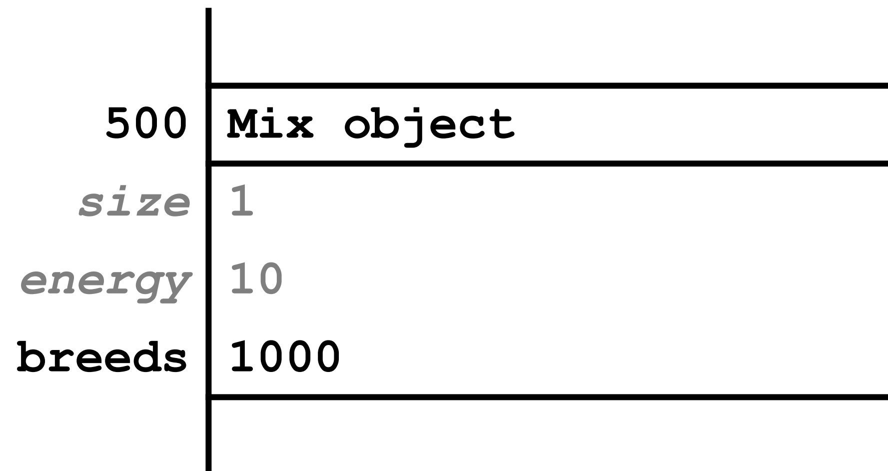
- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
- inheritance does more than copy the API of the superclass
  - the derived class contains a subobject of the parent class
  - the superclass subobject needs to be constructed (just like a regular object)
    - the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

What is a Subclass?

- another model of inheritance is to imagine that the subclass contains all of the fields of the parent class (including the private fields), but cannot directly use the private fields

Mix Memory Diagram

- belongs to superclass
- private in superclass
- not accessible by name to Mix



Constructors of Subclasses

- the purpose of a constructor is to set the values of the fields of **`this`** object
- how can a constructor set the value of a field that belongs to the superclass?
  - by calling the superclass constructor and passing **`this`** as an implicit argument

Constructors of Subclasses

1. the first line in the body of every constructor ***must*** be a call to another constructor
   - if it is not then Java will insert a call to the superclass default constructor
     - if the superclass default constructor does not exist or is private then a compilation error occurs
2. a call to another constructor can only occur on the first line in the body of a constructor
3. the superclass constructor must be called during construction of the derived class

Mix (version 1)

```
public final class Mix extends Dog {
  // no declaration of size or energy; part of Dog
  private ArrayList<String> breeds;

  public Mix () {
    // call to a Dog constructor
    super();
    this.breeds = new ArrayList<String>();
  }

  public Mix(int size, int energy) {
    // call to a Dog constructor
    super(size, energy);
    this.breeds = new ArrayList<String>();
  }
```

```
public Mix(int size, int energy,
           ArrayList<String> breeds) {
  // call to a Dog constructor
  super(size, energy);
  this.breeds = new ArrayList<String>(breeds);
}
```

Mix (version 2 using chaining)

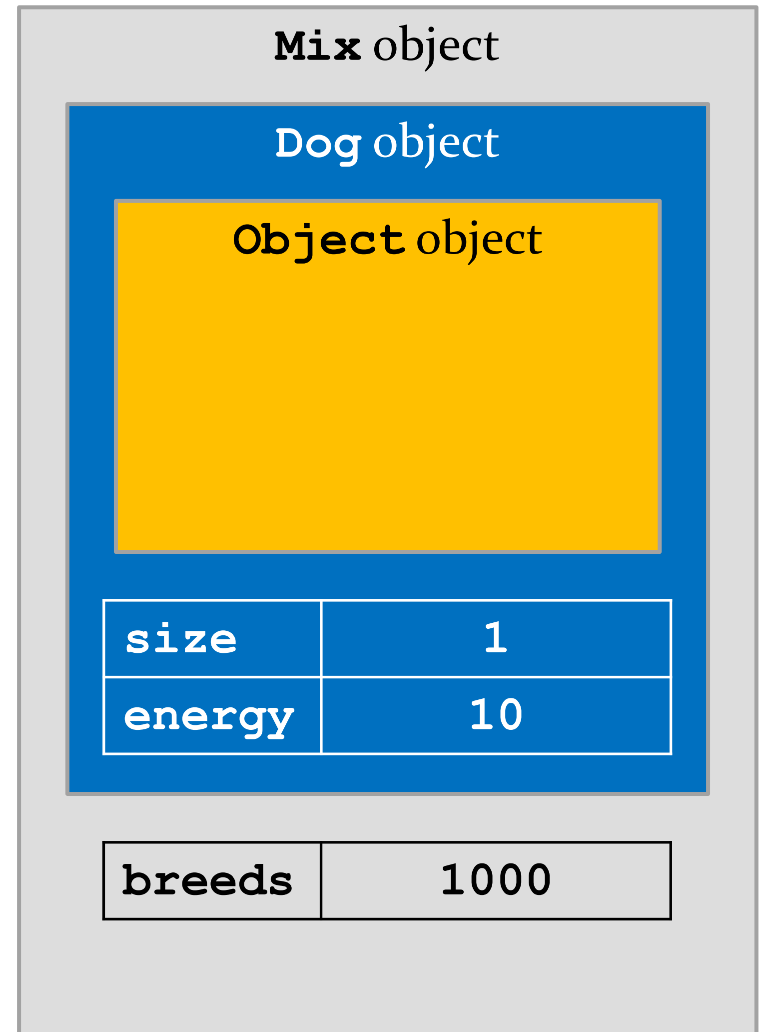
```
public final class Mix extends Dog {
  // no declaration of size or energy; part of Dog
  private ArrayList<String> breeds;

  public Mix () {
    // call to a Mix constructor
    this(5, 5);
  }

  public Mix(int size, int energy) {
    // call to a Mix constructor
    this(size, energy, new ArrayList<String>());
  }
```

```java
public Mix(int size, int energy,
           ArrayList<String> breeds) {
  // call to a Dog constructor
  super(size, energy);
  this.breeds = new ArrayList<String>(breeds);
}
```

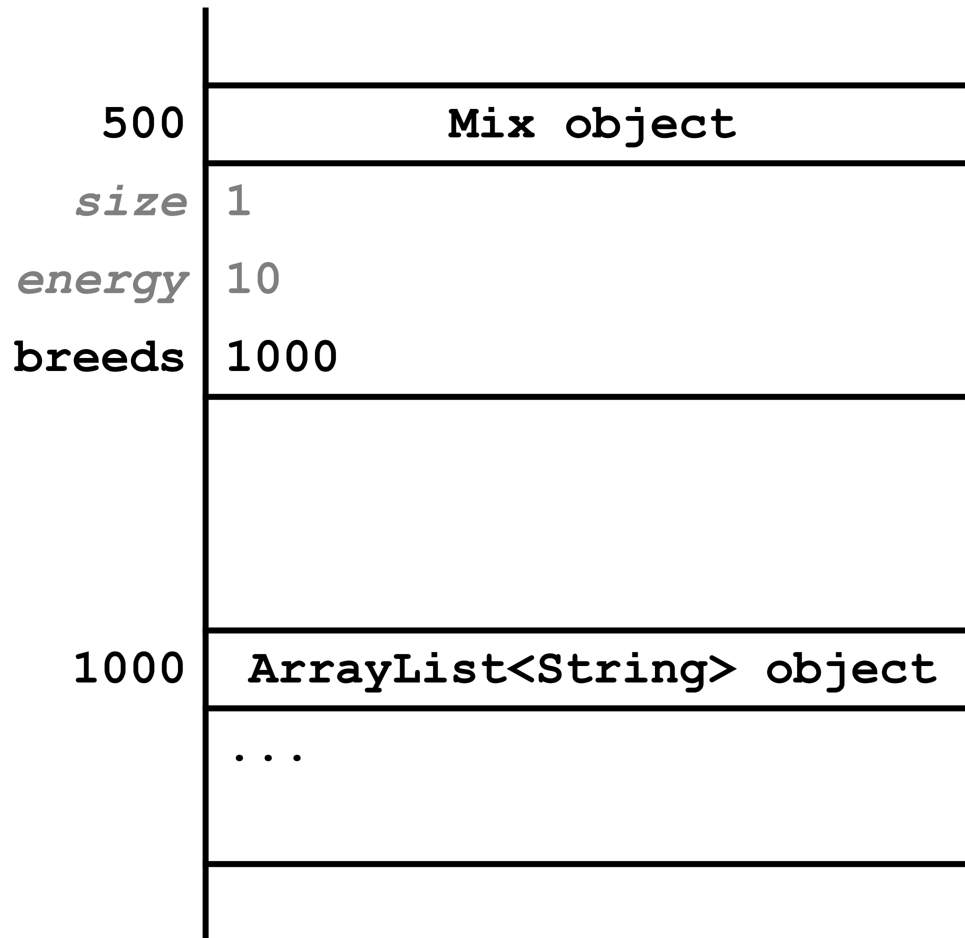
- why is the constructor call to the superclass needed?
  - because **`Mix`** is-a **`Dog`** and the **`Dog`** part of **`Mix`** needs to be constructed

```
Mix mutt = new Mix(1, 10);
```

1. **`Mix`** constructor starts running
- creates new **`Dog`** subobject by invoking the **`Dog`** constructor
    2. **`Dog`** constructor starts running
    - creates new **`Object`** subobject by (silently) invoking the **`Object`** constructor
        3. **`Object`** constructor runs
    - sets **`size`** and **`energy`**
- creates a new empty **`ArrayList`** and assigns it to **`breeds`**



Mix Memory Diagram



Invoking the Superclass Ctor

- why is the constructor call to the superclass needed?
  - because **`Mix`** is-a **`Dog`** and the **`Dog`** part of **`Mix`** needs to be constructed
    - similarly, the **`Object`** part of **`Dog`** needs to be constructed

Invoking the Superclass Ctor

- a derived class can only call its own constructors or the constructors of its immediate superclass
  - **`Mix`** can call **`Mix`** constructors or **`Dog`** constructors
  - **`Mix`** cannot call the **`Object`** constructor
    - **`Object`** is not the immediate superclass of **`Mix`**
  - **`Mix`** cannot call **`PureBreed`** constructors
    - cannot call constructors across the inheritance hierarchy
  - **`PureBreed`** cannot call **`Komondor`** constructors
    - cannot call subclass constructors

Constructors & Overridable Methods

- if a class is intended to be extended then its constructor must not call an overridable method
  - Java does not enforce this guideline
- why?
  - recall that a derived class object has inside of it an object of the superclass
  - the superclass object is always constructed first, then the subclass constructor completes construction of the subclass object
  - the superclass constructor will call the overridden version of the method (the subclass version) even though the subclass object has not yet been constructed

Superclass Ctor & Overridable Method

```
public class SuperDuper {
  public SuperDuper() {
    // call to an over-ridable method; bad
    this.overrideMe();
  }

  public void overrideMe() {
    System.out.println("SuperDuper overrideMe");
  }
}
```

Subclass Overrides Method

```
public class SubbyDubby extends SuperDuper {
  private final Date date;

  public SubbyDubby() {
    super();
    this.date = new Date();
  }

  @Override
  public void overrideMe() {
     System.out.println("SubbyDubby overrideMe : " + this.date);
  }

  public static void main(String[] args) {
     SubbyDubby sub = new SubbyDubby();
     sub.overrideMe();
  }
}
```

- the programmer's intent was probably to have the program print:

```
SuperDuper overrideMe
SubbyDubby overrideMe : <the date>
```

or, if the call to the overridden method was intentional

```
SubbyDubby overrideMe : <the date>
SubbyDubby overrideMe : <the date>
```

- but the program prints:

```
SubbyDubby overrideMe : null
SubbyDubby overrideMe : <the date>
```

final attribute in two different states!

What's Going On?

1. **new SubbyDubby()** calls the **SubbyDubby** constructor
2. the **SubbyDubby** constructor calls the **SuperDuper** constructor
3. the **SuperDuper** constructor calls the method **overrideMe** which is overridden by **SubbyDubby**
4. the **SubbyDubby** version of **overrideMe** prints the **SubbyDubby date** field which has not yet been assigned to by the **SubbyDubby** constructor (so **date** is null)
5. the **SubbyDubby** constructor assigns **date**
6. **SubbyDubby overrideMe** is called by the client

- remember to make sure that your base class constructors only call **`final`** methods or **`private`** methods
  - if a base class constructor calls an overridden method, the method will run in an unconstructed derived class