TA Session 6

Overriding vs shadowing

Late vs early binding

Up and down casting

Overriding

```
public class Person {
    public String chitchat() {
        return "Nice weather!";
    }
}

public class Student extends Person {
    public String chitchat() {
        return "Did you finish ex2?";
    }
}
```

Overriding

```
public static void main(String[] cmdArgs) {

   Student stud = new Student();
   Person person = stud;

   System.out.println(   stud.chitchat() );
   System.out.println( person.chitchat() );
}
```

```
Did you finish ex2?
Did you finish ex2?
```

```
public class Person {
    public int sleepDuration = 8;
}
```

Don’t do this at home

```
public class Student extends Person {
    public int sleepDuration = 7;
}
```

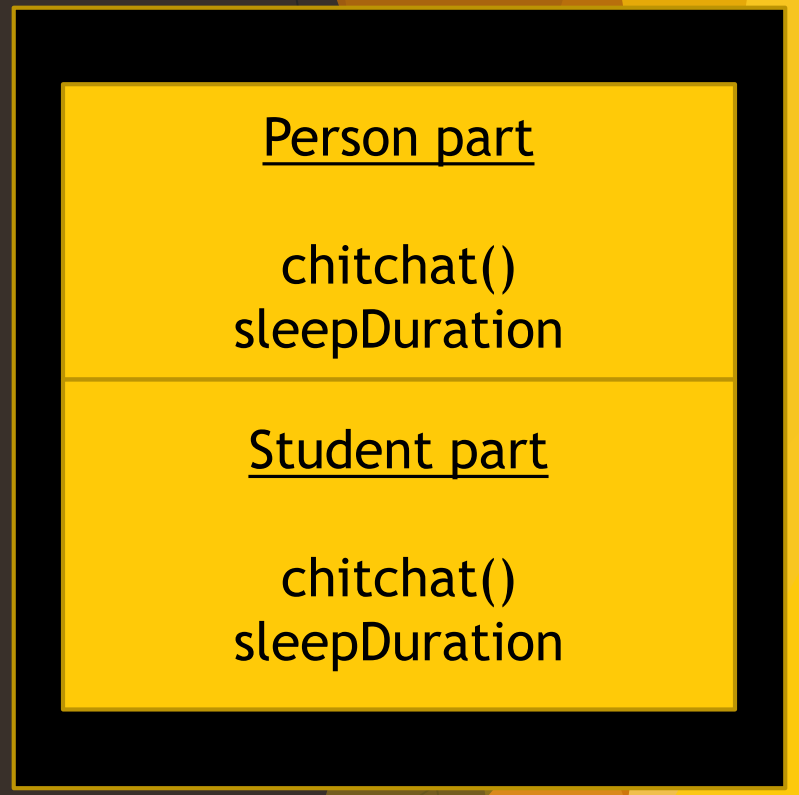
```
public static void main(String[] cmdArgs) {

    Student stud = new Student();
    Person person = stud;

    System.out.println("duration: "+  stud.sleepDuration);
    System.out.println("duration: "+person.sleepDuration);
}
```

duration: 7
duration: 8

What what what??

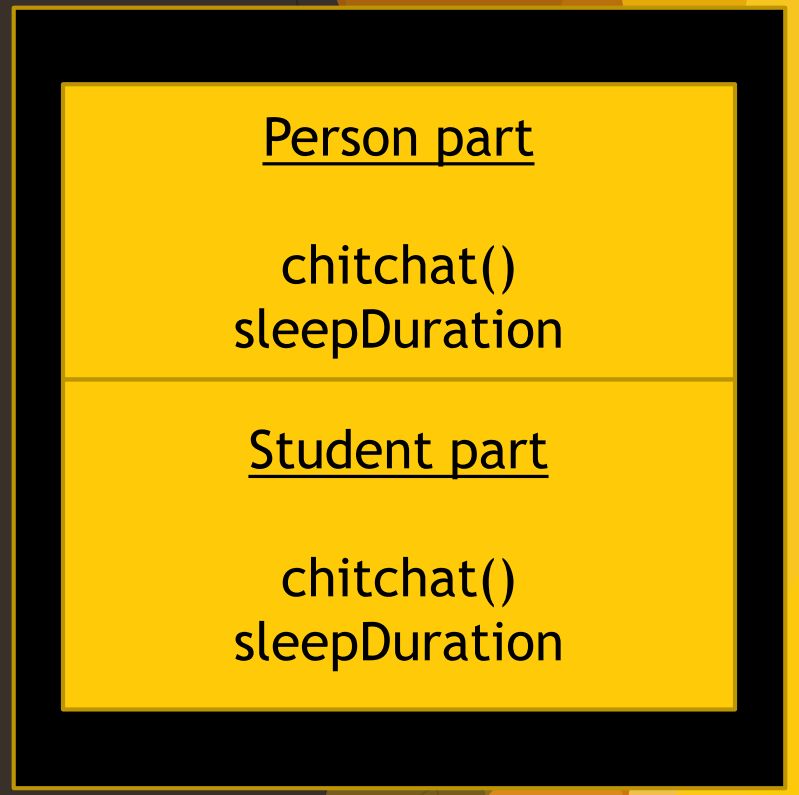
Shadowing

- Superclass members are never discarded!
- It’s just a matter or accessing them.



Shadowing

- 4 types of members:
  - Methods
  - Fields
  - Static methods
  - Static fields



Shadowing

- 4 types of members:
  - Methods → Always most specific version
  - Fields
  - Static methods
  - Static fields

  } According to reference-type!

Shadowing

```
Student stud = new Student();
```

Is there ANY way to call Person's chitchat??

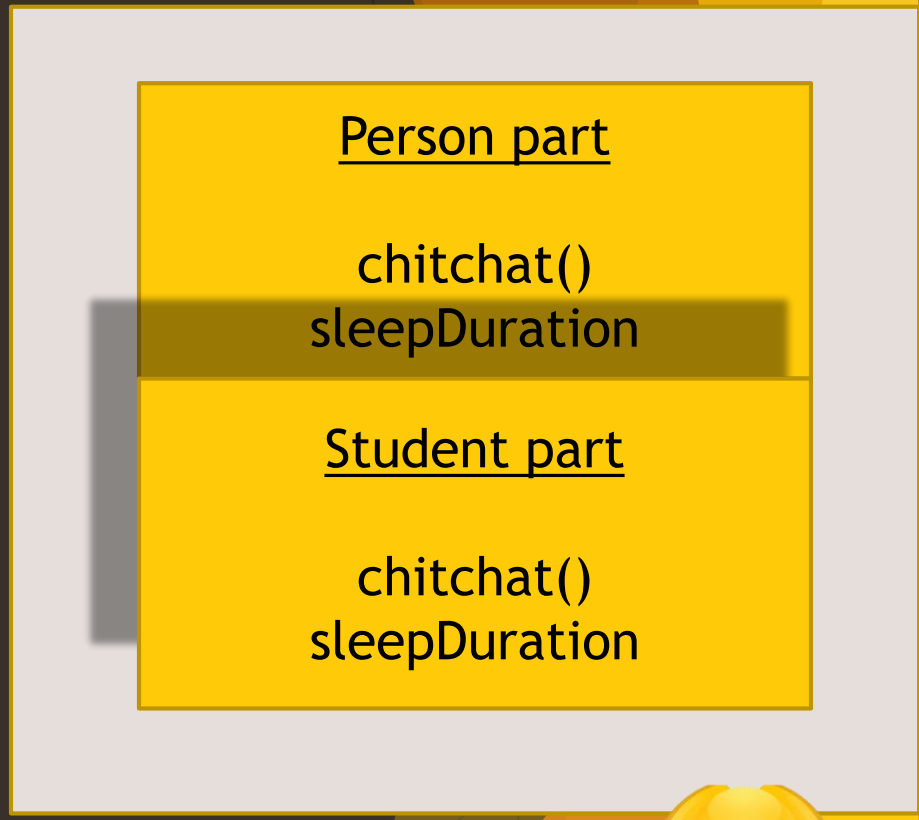
Shadowing

- If a superclass member is accessible using a superclass reference,

  We say a subclass member with the same declaration only ***shadows*** it,

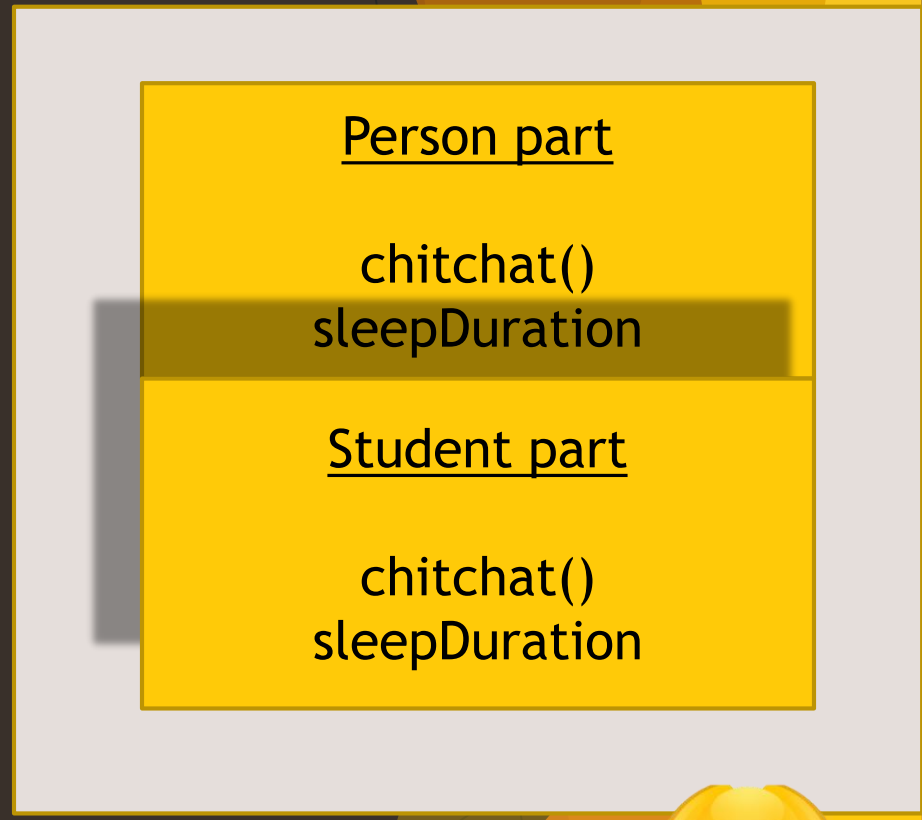
  or ***hides*** it, not overrides it.

- Shadowed members are usually not what we want, but still accessible from outside.



Shadowing

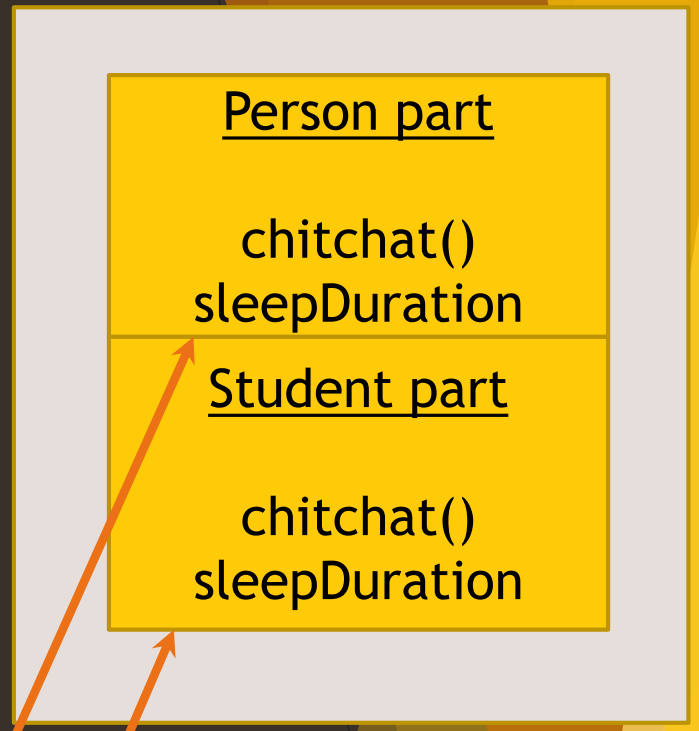
- How can shadowed members be accessed?
  - Superclass reference
  - *Super* keyword, from within the class



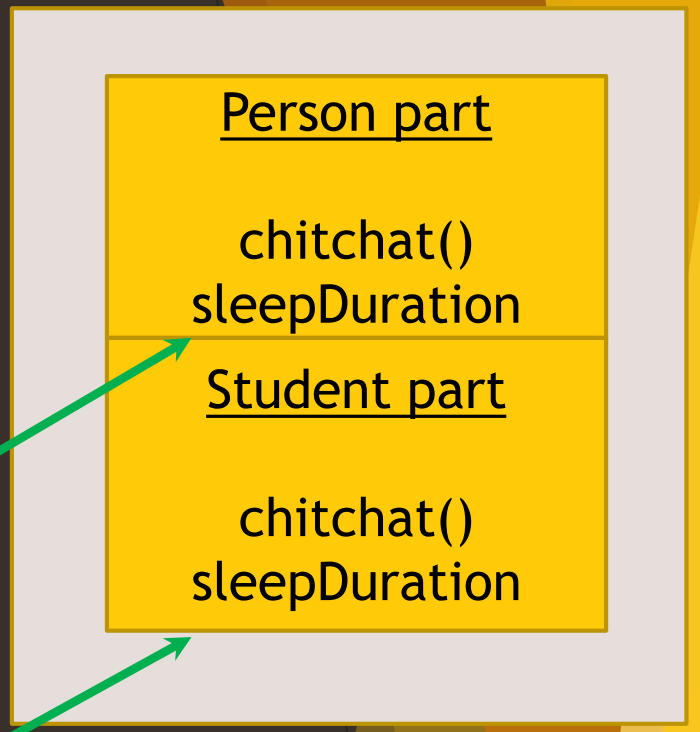
```
public class Person {
   public int sleepDuration = 8;
}
```

```
public class Student extends Person {
   public int sleepDuration = 7;
}
```

```
public static void main(String[] cmdArgs) {

    Student stud = new Student();
    Person person = stud;

    System.out.println("duration: "+  stud.sleepDuration);
    System.out.println("duration: "+person.sleepDuration);
}
```



```java
public class Person {
   public int sleepDuration = 8;
}


public class Student extends Person {
   public int sleepDuration = 7;

   public String chitChat() {
      return "Did you finish ex2? "+
         "I only slept "+sleepDuration+
         "hours, while you slept "+
         super.sleepDuration;
}

}
```

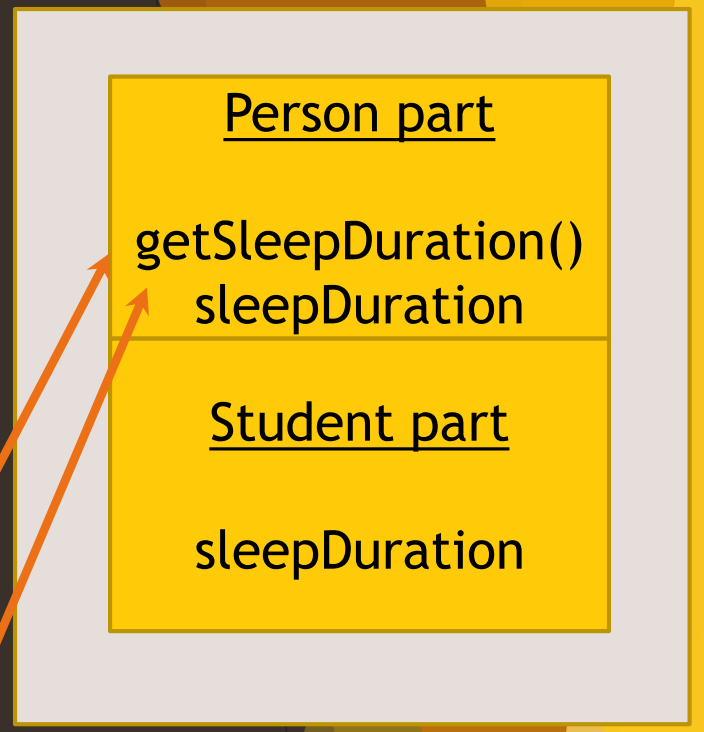


Avoiding shadowed fields: solution!

```
public class Person {
   private int sleepDuration = 8;
   public int getSleepDuration() {
      return sleepDuration;
   }
}

public class Student extends Person {
   private int sleepDuration = 7;
}
```

```
public static void main(String[] cmdArgs) {

    Student stud = new Student();
    Person person = stud;

    System.out.println("duration: "+  stud.getSleepDuration());
    System.out.println("duration: "+person.getSleepDuration());
}
```

```
duration: 8
duration: 8
```

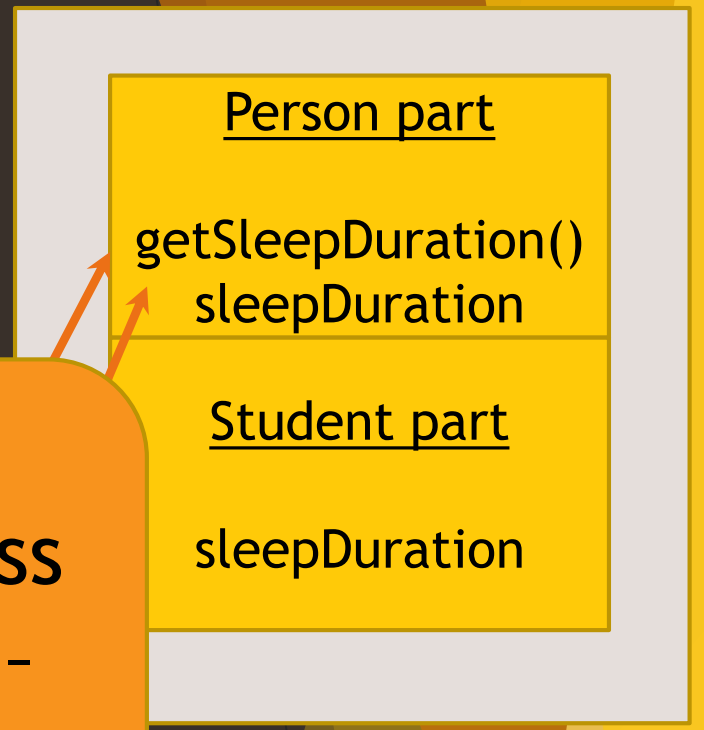


What what what??

```
public stati

    Student s
    Person pe

    System.ou
    System.out
}
```

```
duration: 8
duration: 8
```

Person’s members can only access overriding members of Sutdent - not shadowing members

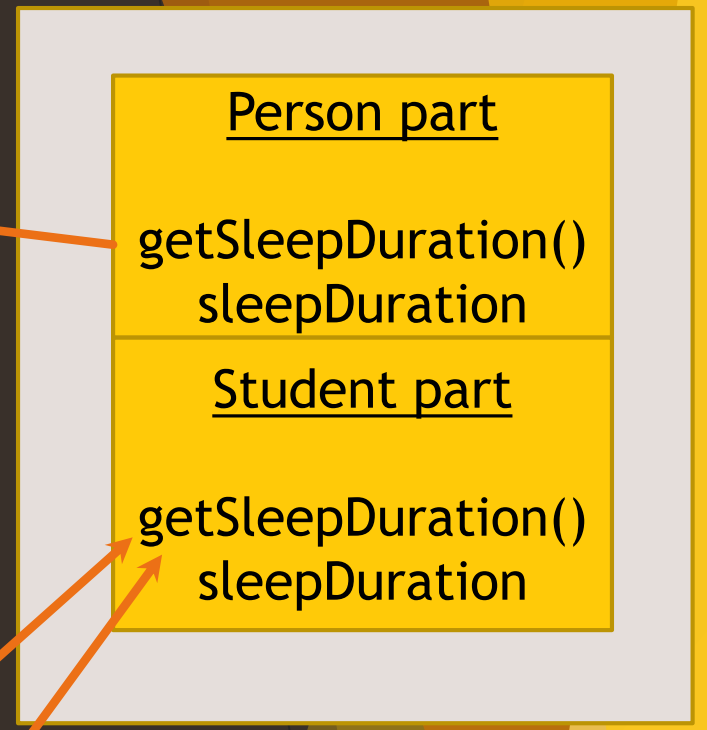


What what what??

Solution!

```
public class Person {
   public int sleepDuration = 8;
   public int getSleepDuration() {
      return sleepDuration;
   }
}

public class Student extends Person {
   public int sleepDuration = 7;
   public int getSleepDuration() {
      return sleepDuration;
   }
}
```

Still here!
How can it be accessed?




```
public static void main(String[] cmdArgs) {

    Student stud = new Student();
    Person person = stud;

    System.out.println("duration: "+  stud.getSleepDuration());
    System.out.println("duration: "+person.getSleepDuration());
}
```

```
duration: 7
duration: 7
```

What about static methods?

```
public class Person {
   public static void evolve() {
      System.out.println(
            "new persons will have super mutations");
      // ..change genome..
   }
}

public class Student extends Person {
   public static void evolve() {
      System.out.println(
            "new students will be super-studs");
      // ..make students like this
   }
}
```

```
public static void main(String[] cmdArgs) {

   Student stud = new Student();
   Person person = stud;

   stud.evolve());
   person.evolve());
}
```

new students will be super-studs
new persons will have super mutations

Shadowing

- Accessing static methods via class (not instance) prevents confusion.

- But why? Why shadow??

Late binding

- Late, or run-time binding is *postponing the resolving of a method address to run-time*

```
person.chitchat();
```


```
mov eax, ebx
add ebx, ecx
call ???
```

Late binding

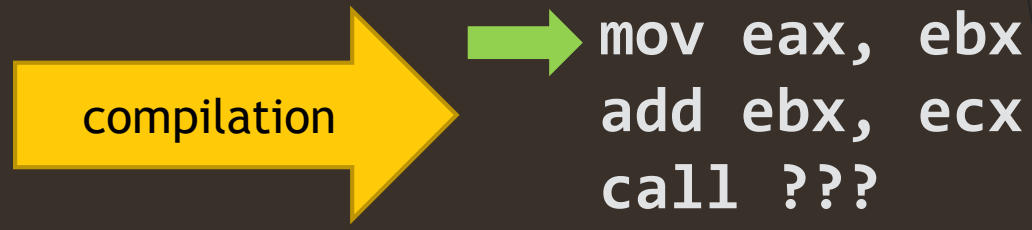
- Only the actual instance holds the correct address!
- How would they represent the information?

```
person.chitchat();
```


```
mov eax, ebx
add ebx, ecx
call ???
```

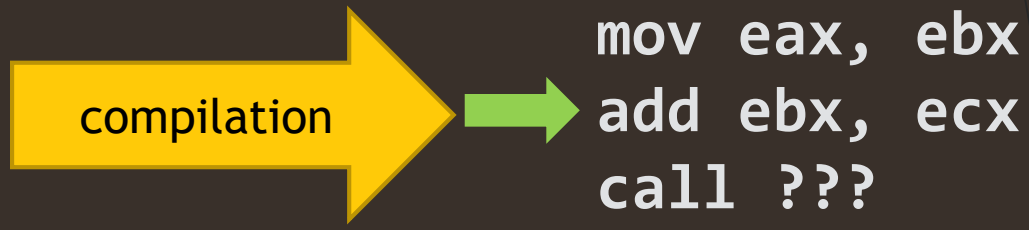
At run-time:

```
person.chitchat();
```



At run-time:

```
person.chitchat();
```



At run-time:

```
person.chitchat();
```

compilation

```
mov eax, ebx
add ebx, ecx
call ???
```

person

Method name	Address
chitchat	0x006ab274
getSleepDuration	0x006ab290

Person part

Student part

At run-time:

```
person.chitchat();
```

compilation

```
mov eax, ebx
add ebx, ecx
call 0x006ab274
```

person

Method name	Address
chitchat	0x006ab274
getSleepDuration	0x006ab290

Person part

Student part

Late binding

- Takes more time
- Takes more memory

Early binding

- Early, or compile-time binding is inserting the address at compilation…
- According to reference type.

```
person.evolve();
```


```
mov eax, ebx
add ebx, ecx
call 0x00724ba2
```

Early binding

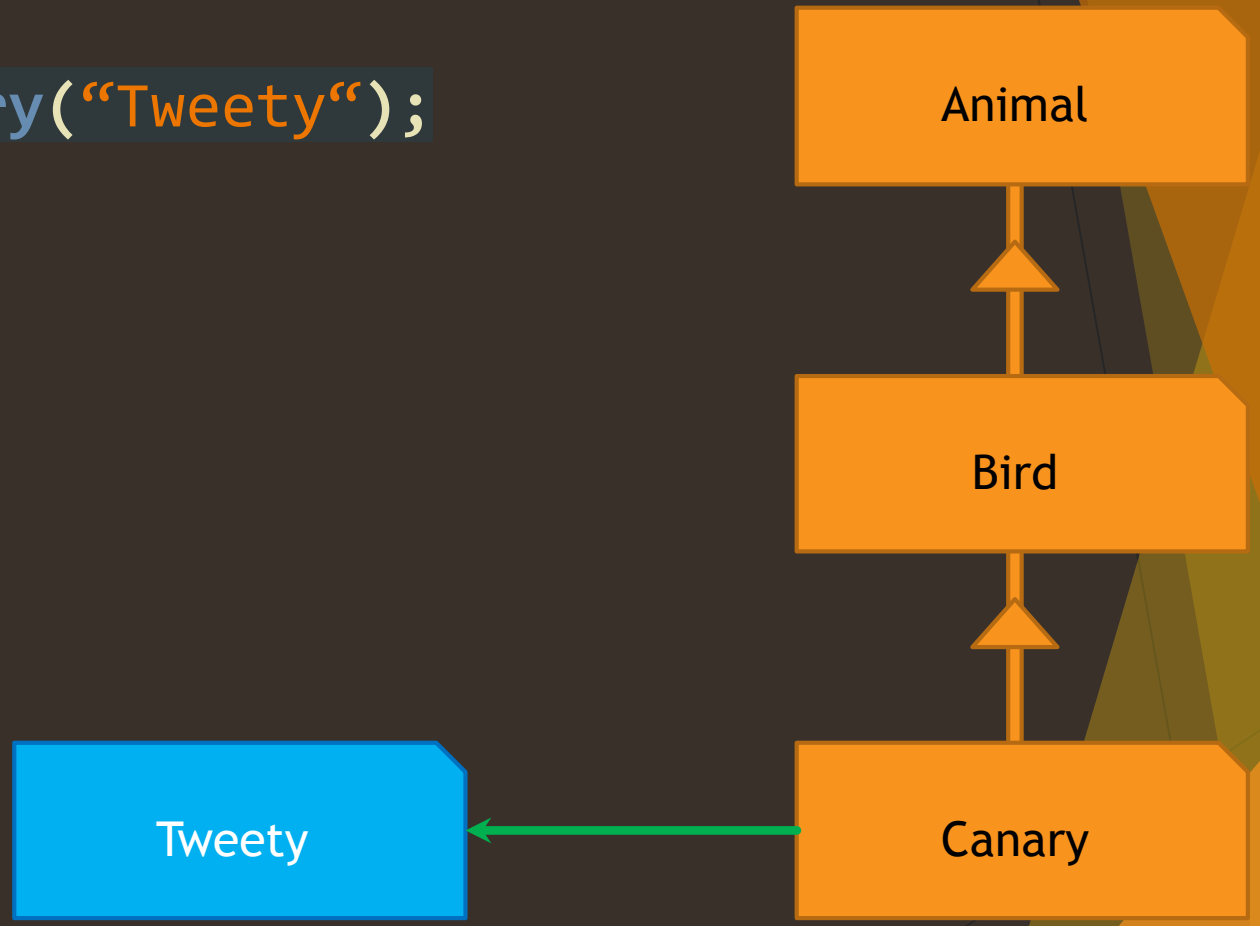
- Is more efficient, less intuitive.
- That much more efficient? Really?

- No.
- For C++ inventors, it was.
- Early binding is default in C++ !
- Java made late binding default for methods…
- But otherwise conforms.

Casting:
up and down

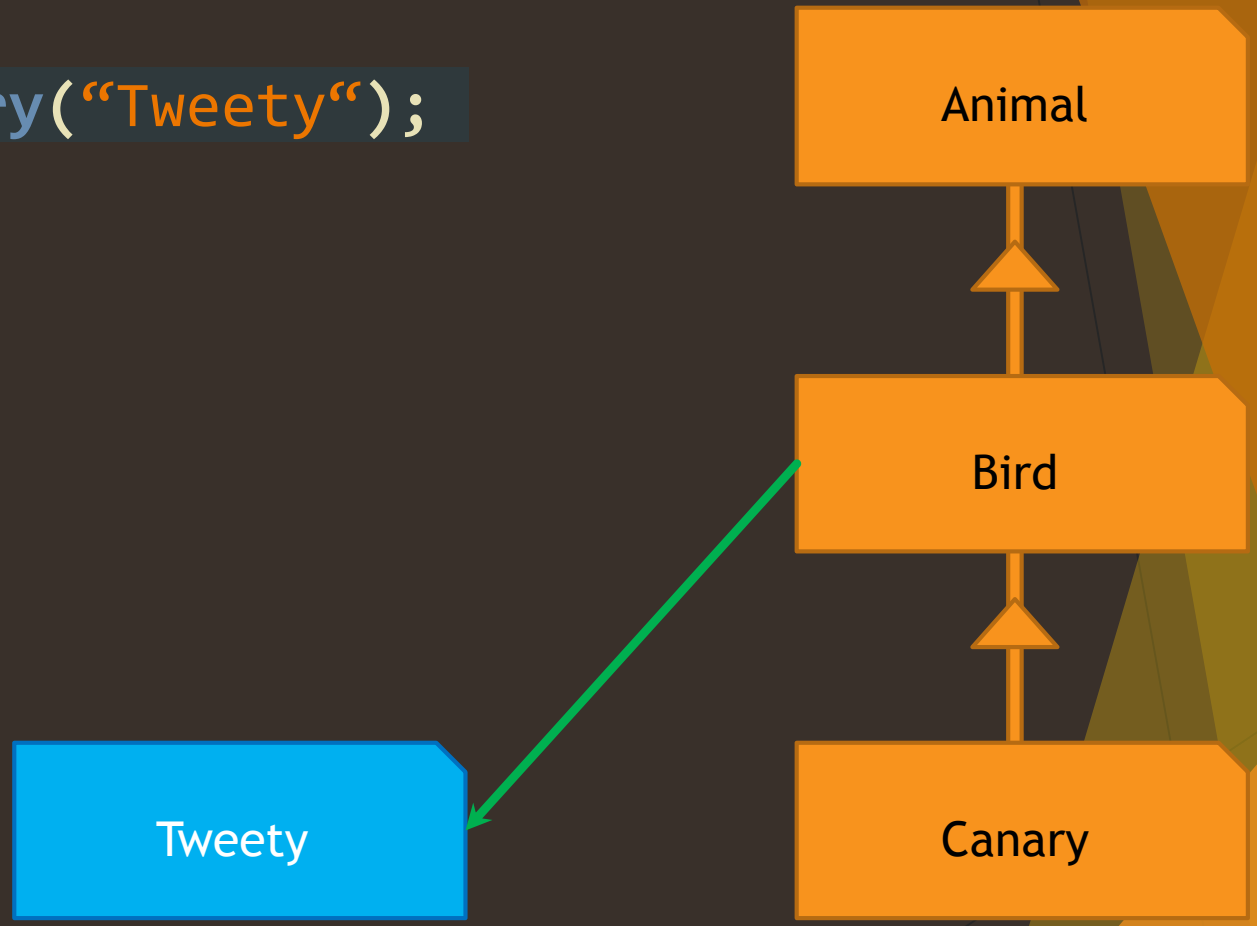
Up-casting

```
Canary tweety = new Canary("Tweety");
```



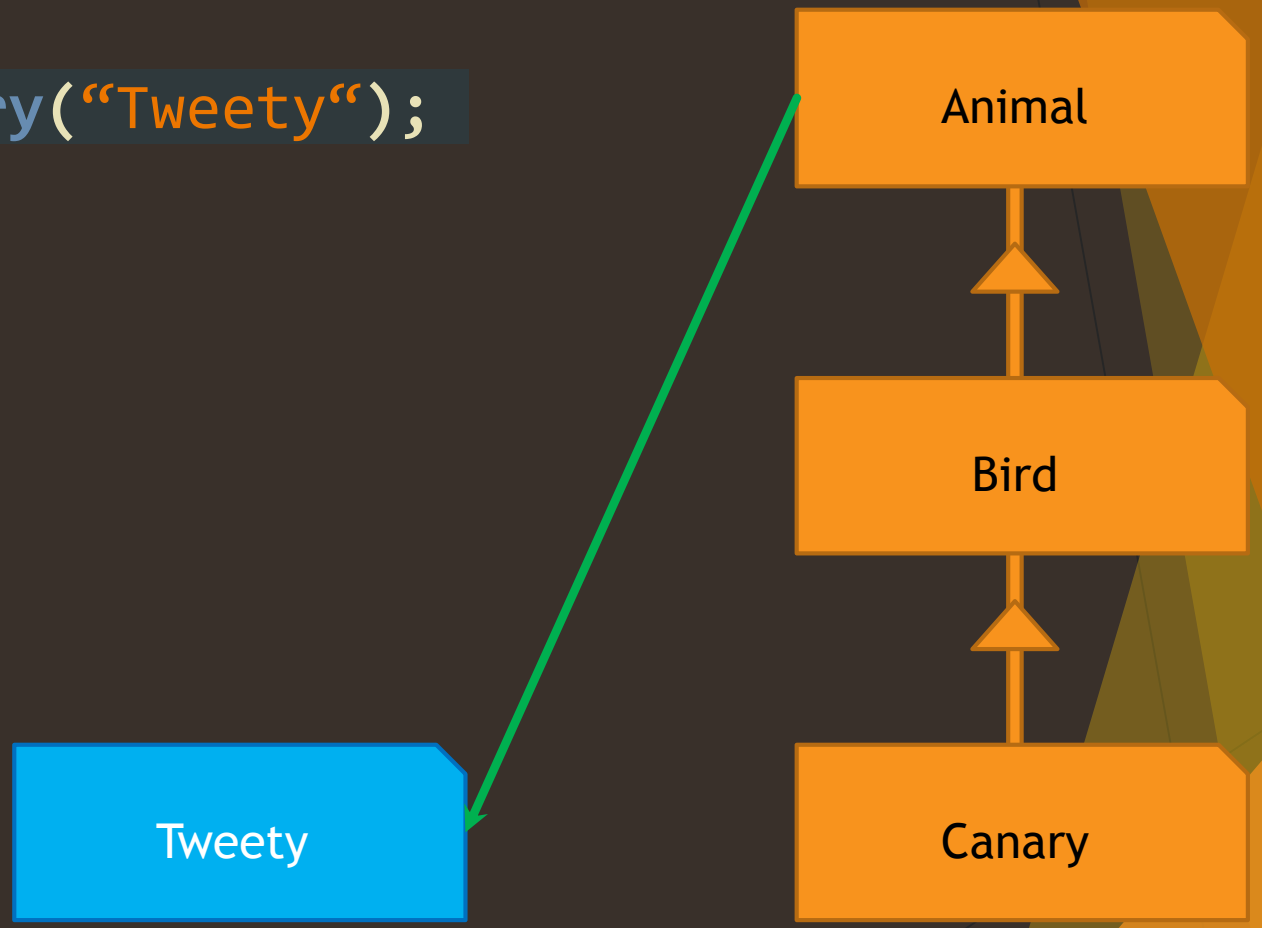
Up-casting

```
Canary tweety = new Canary(“Tweety“);
Bird b = tweety;
```



Up-casting

```
Canary tweety = new Canary(“Tweety“);
Bird b = tweety;
Animal a = b;
```



Down-casting

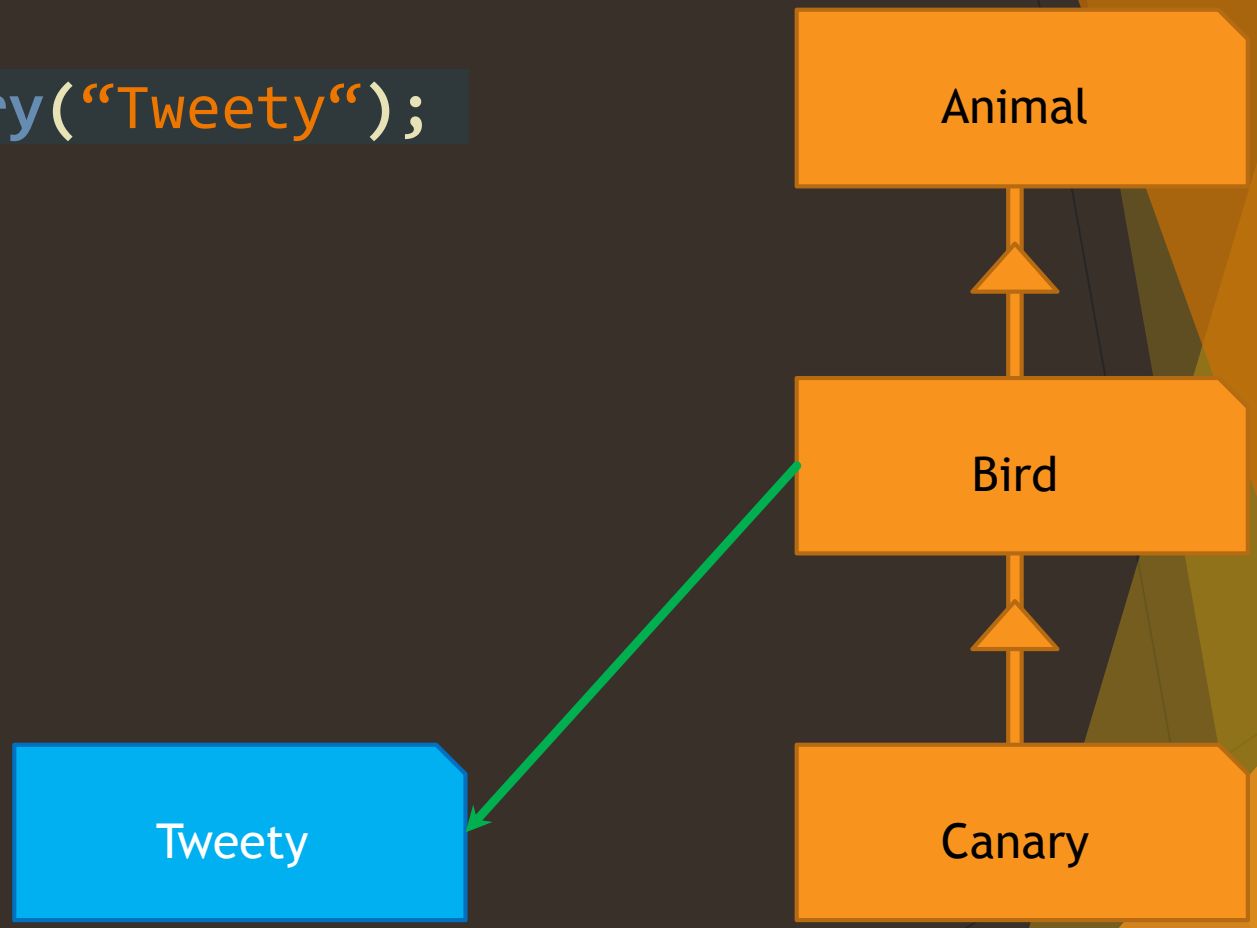
```
Canary tweety = new Canary("Tweety");
Bird b = tweety;
Animal a = b;
b = a;
```

Hold on mister buddy…
You sure ‘a’ is a Bird?

Down-casting

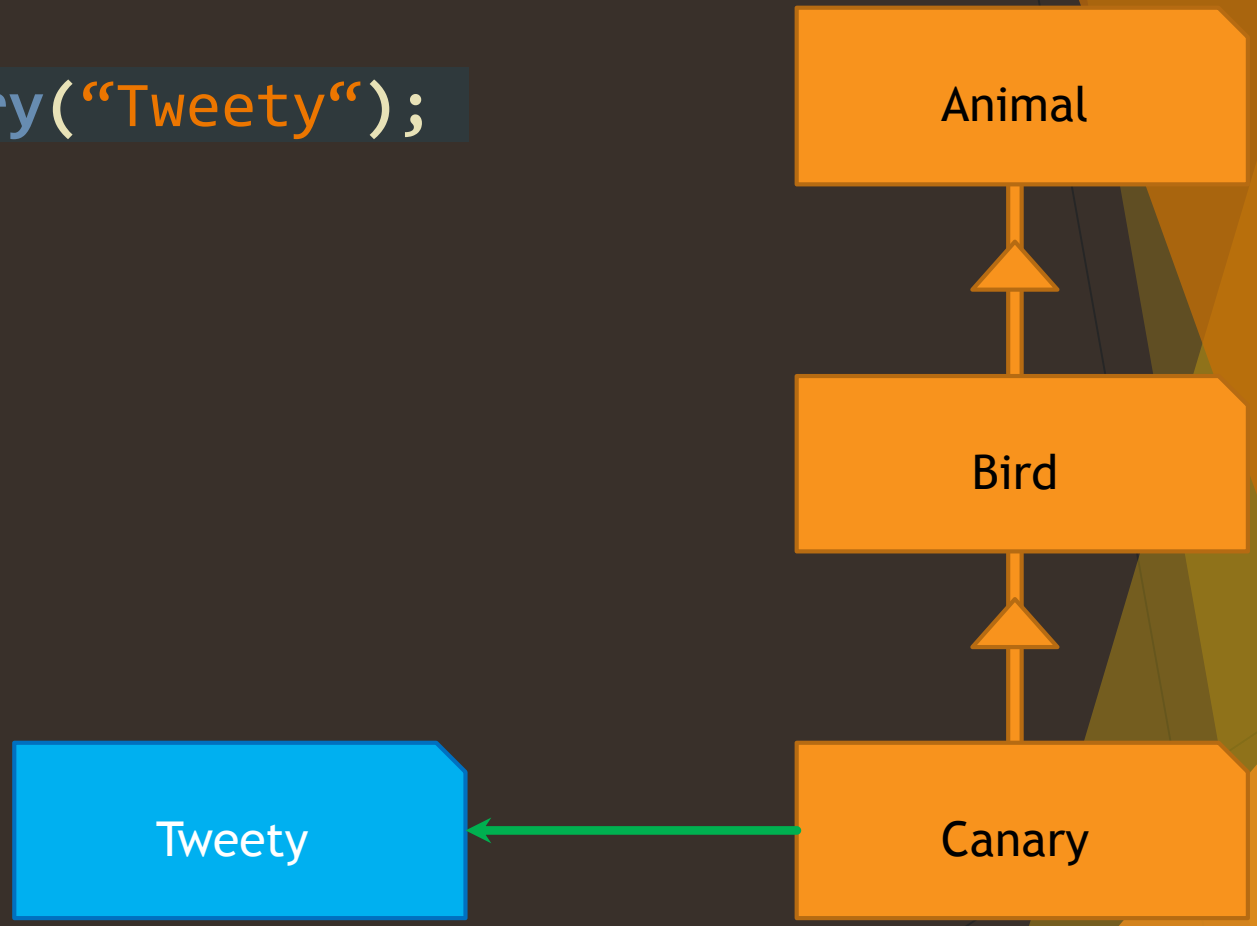
```
Canary tweety = new Canary(“Tweety“);
Bird b = tweety;
Animal a = b;
b = (Bird)a;
```

Now it’s **our** responsibility



Down-casting

```
Canary tweety = new Canary(“Tweety“);
Bird b = tweety;
Animal a = b;
b = (Bird)a;
Canary c = (Canary)a;
```



Down-casting

- Sometimes we don’t know if ‘bird’ is a Canary or a Duck !

Instanceof operator

Down-casting

- Sometimes we don’t know if ‘bird’ is a Canary or a Duck !

```
for(Bird bird : birdcage) {
   if(bird instanceof Canary)
      ((Canary)bird).tweet();
   if(bird instanceof Duck)
      ((Duck)bird).quack();
}
```

instanceof

- Has its place.
- But is time consuming!

- Not too much…

instanceof

- But it smells bad: something might be wrong with your design.

```
for(Bird bird : birdcage) {
   if(bird instanceof Canary)
      bird.tweet();
   if(bird instanceof Duck)
      bird.quack();
}
```

instanceof

- But it smells bad: something might be wrong with your design.

```
for(Bird bird : birdcage) {
   bird.makeSound();
}
```

Use polymorphism.

instanceof

- But it smells bad: something might be wrong with your design.

```
for(Bird bird : birdcage) {
   bird.makeSound();
}
```

"Anytime you find yourself writing code of the form 'if the object is of type T1, then do something, but if it's of type T2, then do something else,' slap yourself."
(Scott Meyers, "Effective C++")