CS/ENGRD 2110 FALL 2017

Lecture 6: Consequence of type, casting; function equals

http://courses.cs.cornell.edu/cs2110

Overview ref in JavaHyperText

- Quick look at arrays array
- Casting among classes cast
- Operator instanceof
- Function getClass
- Function equals
- compile-time reference rule

Homework. JavaHyperText while-loop for-loop

while (<bool expr>) { … } // syntax

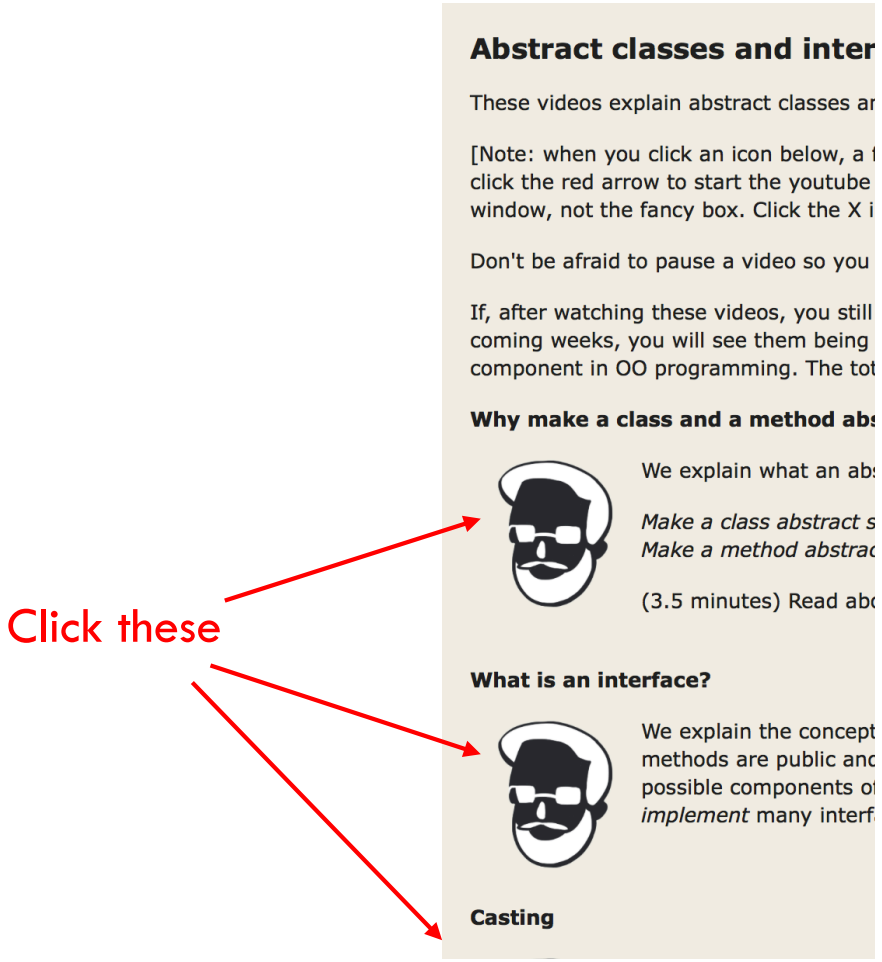
for (**int** k= 0; k < 200; k= k+1) { … } // example

Announcements

- Search Piazza for your question (before posting)!
- Partner-finding event:
  - Tuesday, September 12 at 5:30pm
  - Phillips 203
  - There will be snacks!

Before Next Lecture…

- Follow the tutorial on **abstract classes and interfaces**, and watch the videos.

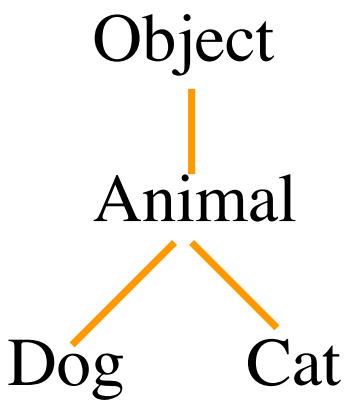


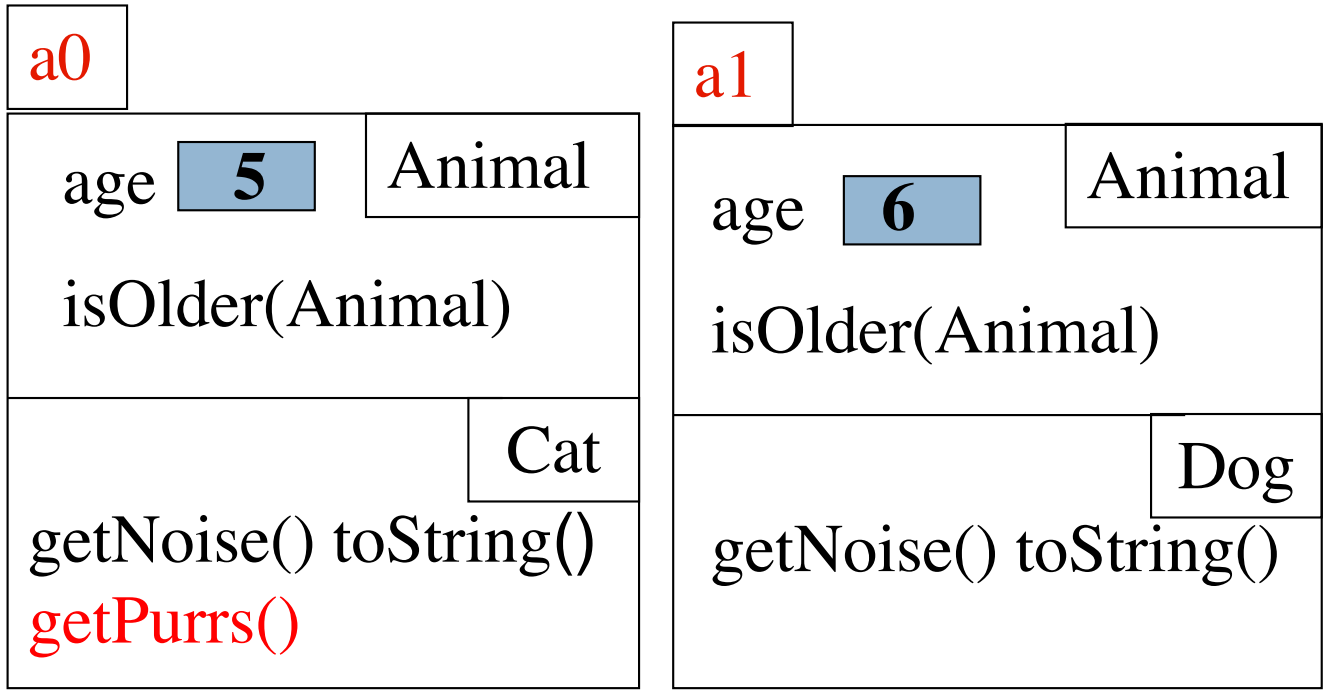
Classes we work with today

Work with a class Animal and subclasses like Cat and Dog

Put components common to animals in Animal

class hierarchy:





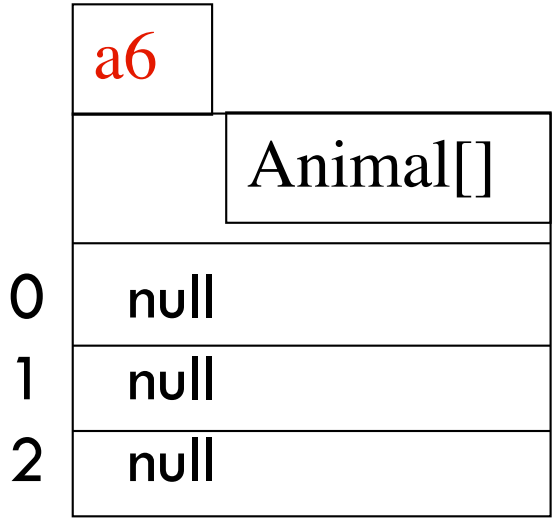
Object partition is there but not shown

Animal[] v= **new** Animal[3];

declaration of array v

Create array of 3 elements

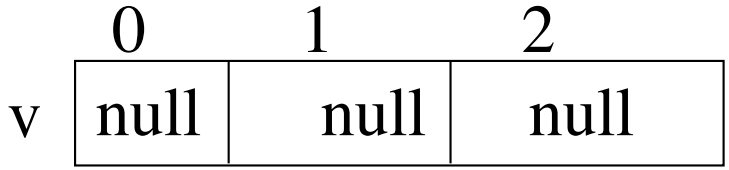
Assign value of new-exp to v



Assign and refer to elements as usual:

```
v[0]= new Animal(…);
…
a= v[0].getAge();
```

Sometimes use horizontal picture of an array:

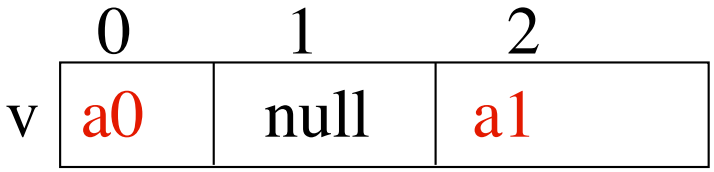


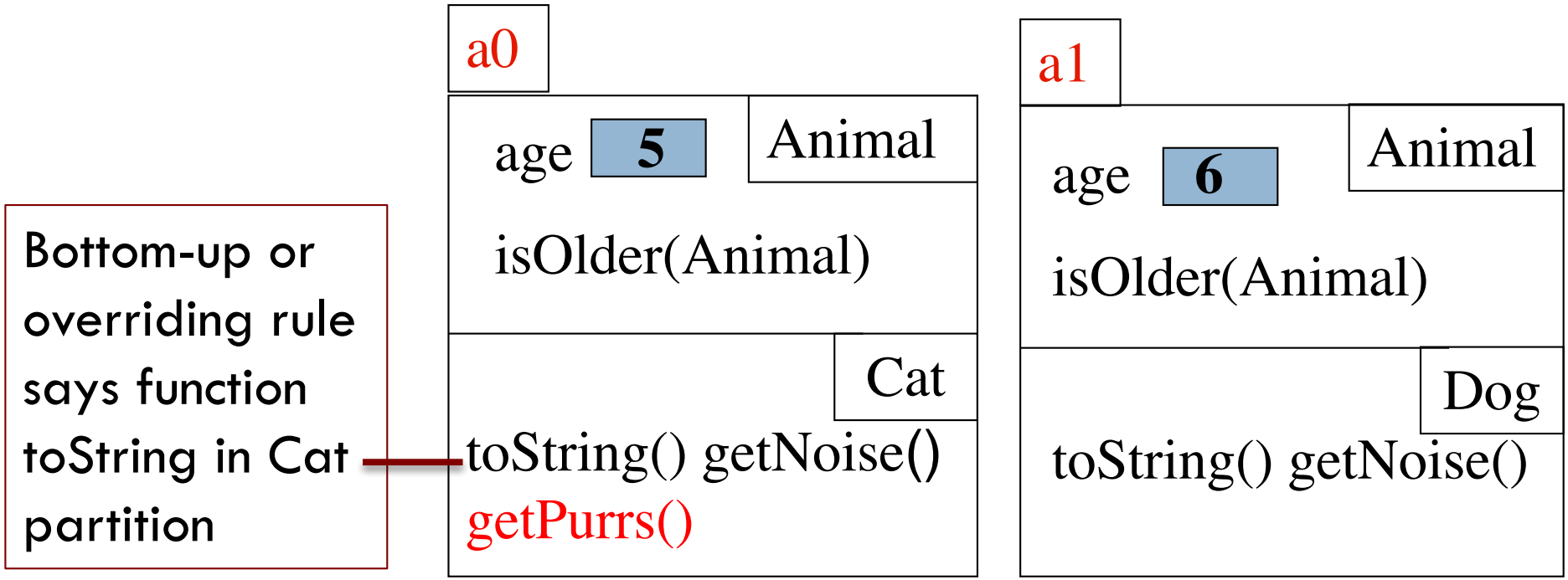
Which function is called?

Which function is called by

v[0].toString() ?

(Remember, the hidden Object partition contains toString().)

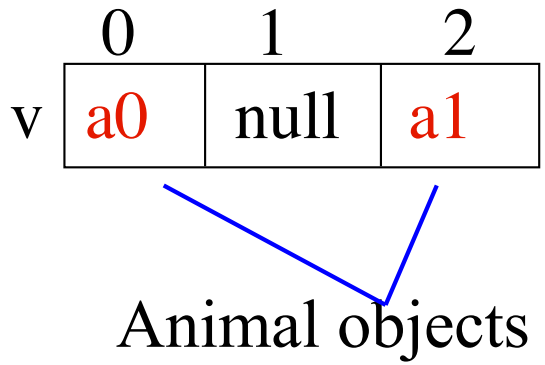




Consequences of a class type

Animal[] v; declaration of v. Also means that each variable v[k] is of type Animal

The type of v is Animal[]
The type of each v[k] is Animal
The type is part of the syntax/grammar of the language. Known at compile time.

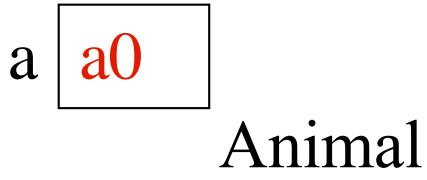


A variable's type:

- *Restricts* what values it can contain.
- Determines which methods are legal to call on it.

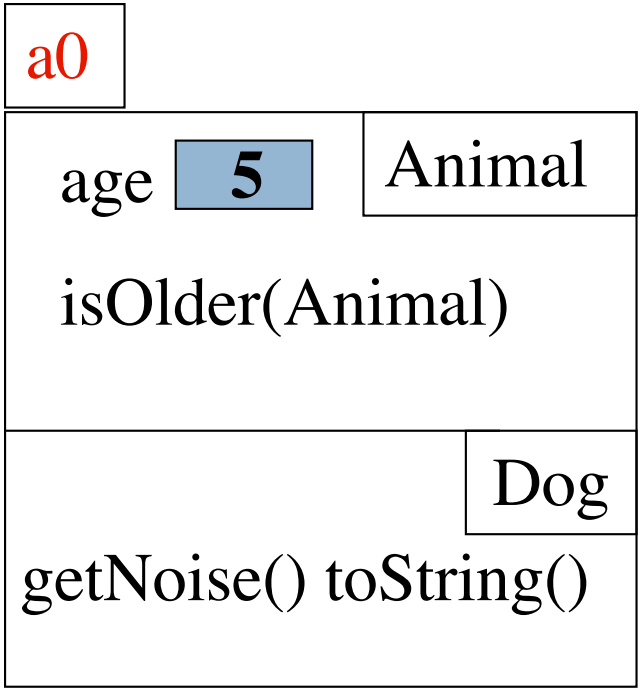
From an Animal variable, can use only methods available in class Animal

a.getPurrs() is obviously illegal.
The compiler will give you an error.



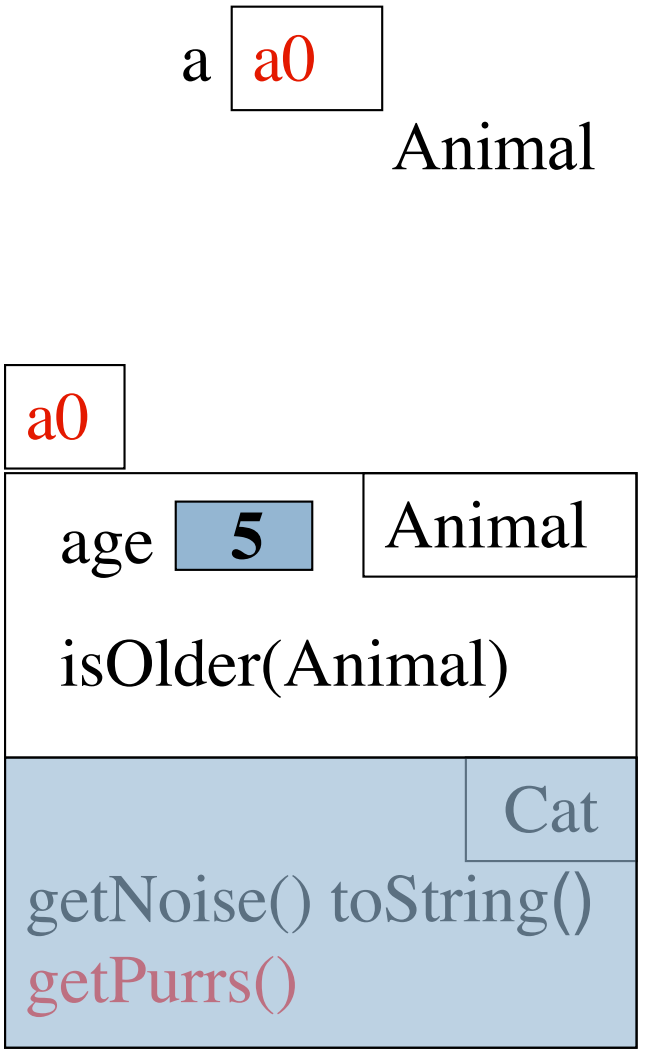
When checking legality of a call like
a.getPurrs(…)
since the type of a is Animal, method getPurrs must be declared in Animal or one of its superclasses.

see JavaHyperText: compile-time reference rule



From an Animal variable, can use only methods available in class Animal

Suppose a0 contains an object of a subclass Cat of Animal. By the rule below, a.getPurrs(…) is still illegal. Remember, the test for legality is done at compile time, not while the program is running.



When checking legality of a call like

a.getPurrs(…)

since the type of a is Animal, method getPurrs must be declared in Animal or one of its superclasses.

see JavaHyperText: compile-time reference rule

From an Animal variable, can use only methods available in class Animal

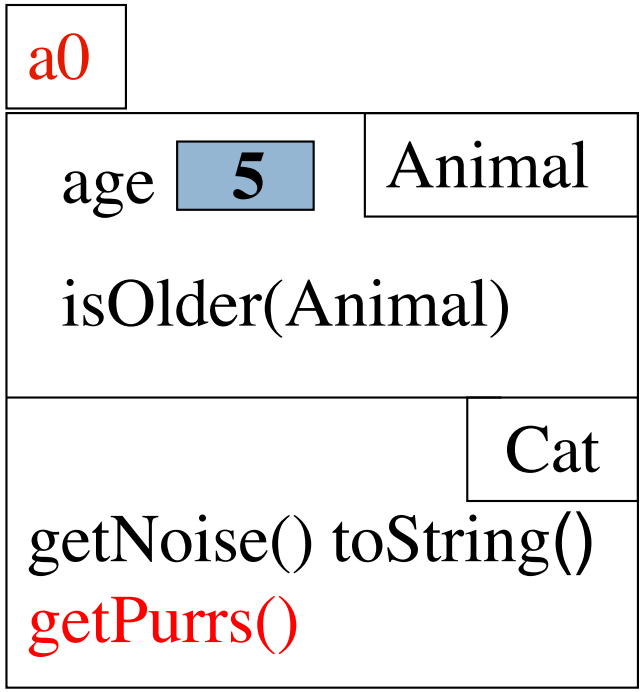
The same object a0, from the viewpoint of a Cat variable and an Animal variable

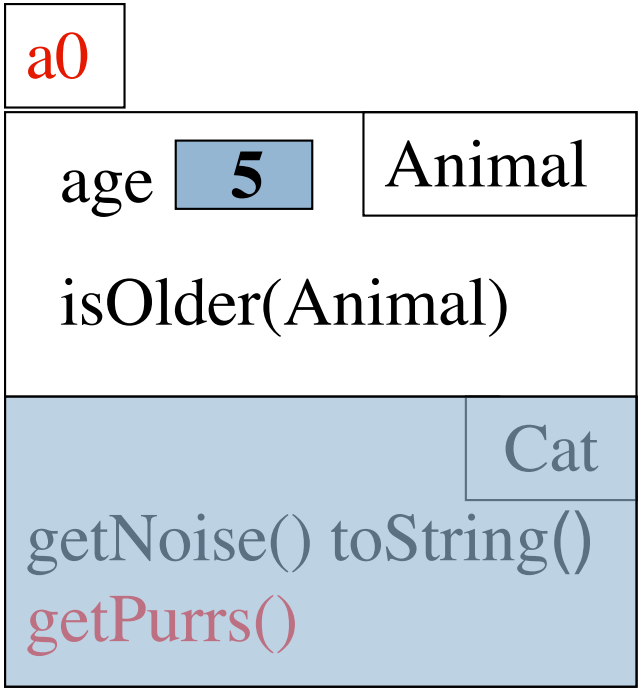
c a0 Cat

a a0 Animal

c.getPurrs() is legal

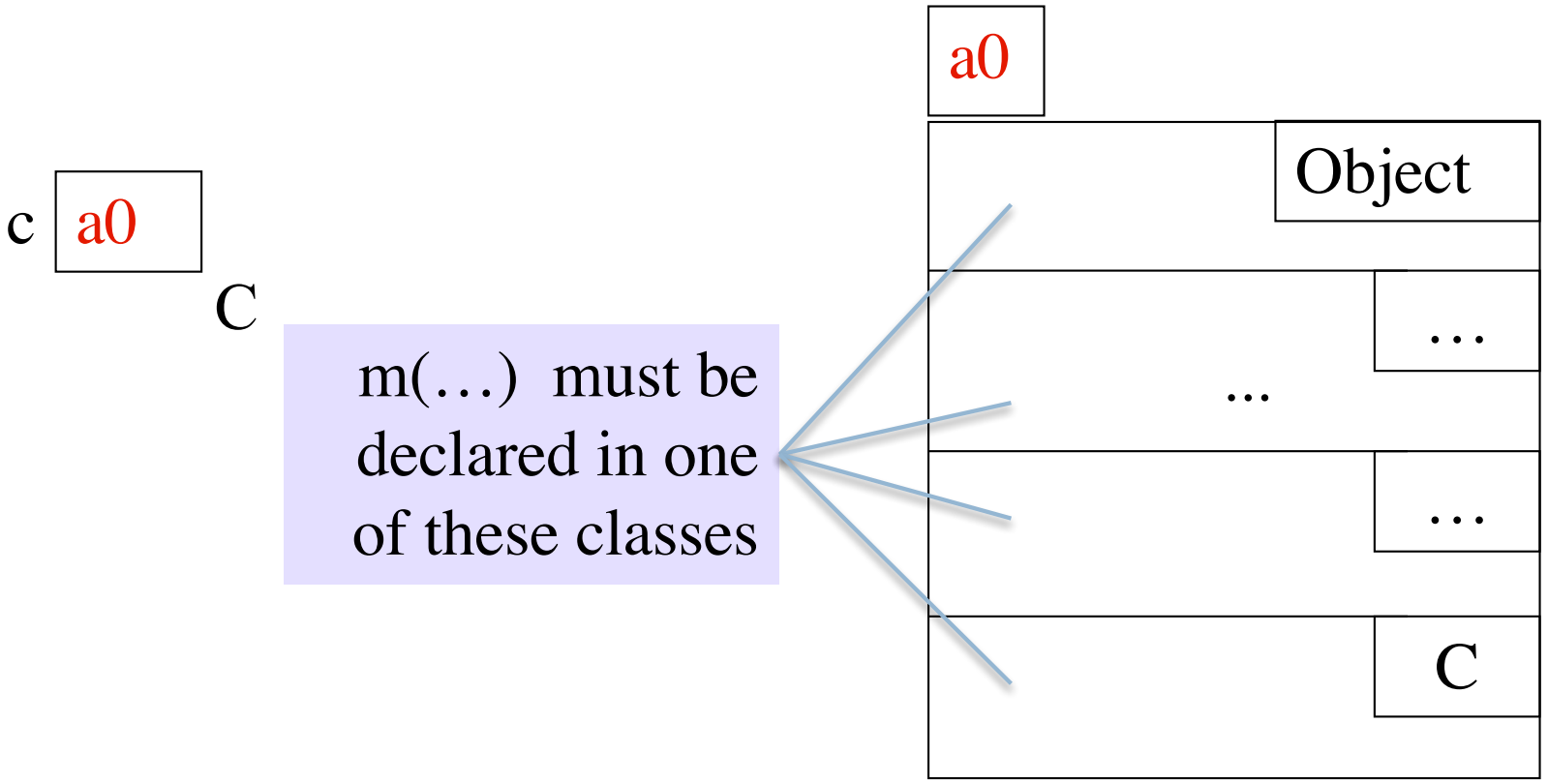
a.getPurrs() is illegal because getPurrs is not available in class Animal





Rule for determining legality of method call

Rule: c.m(…) is legal and the program will compile ONLY if method m is declared in C or one of its superclasses. (JavaHyperText entry: compile-time reference rule.)



Another example

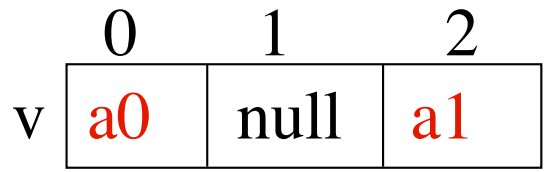
Type of v[0]: Animal

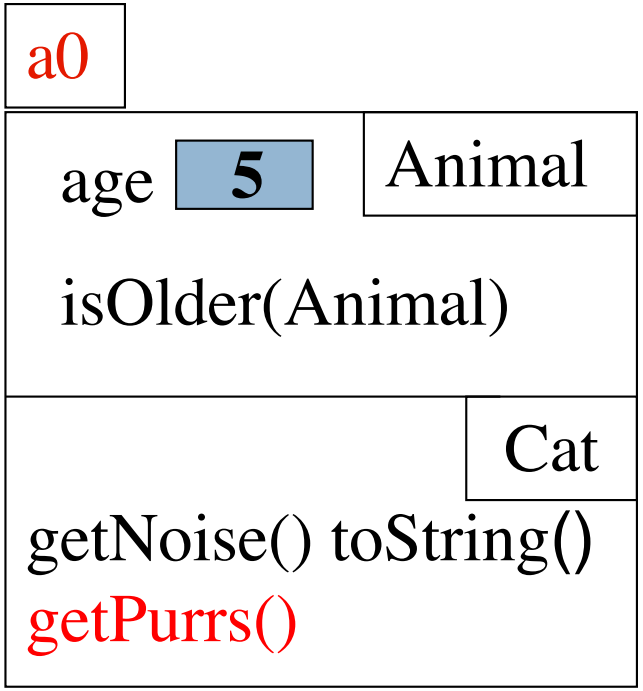
Should this call be allowed?
Should program compile?

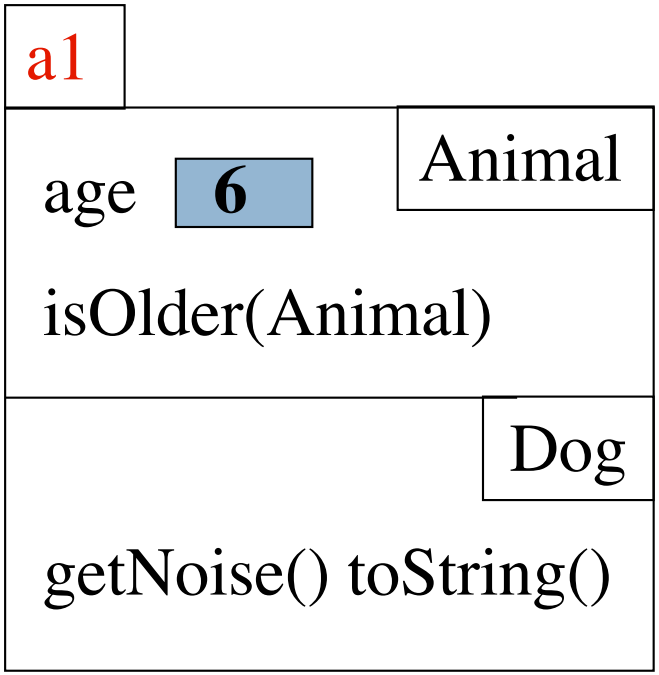
v[k].getPurrs()

Should this call be allowed?
Should program compile?

v[0].getPurrs()







View of object based on the type

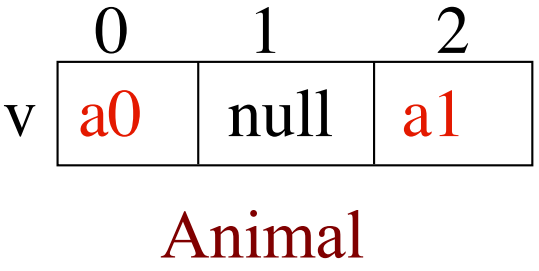
Each element v[k] is of type Animal.

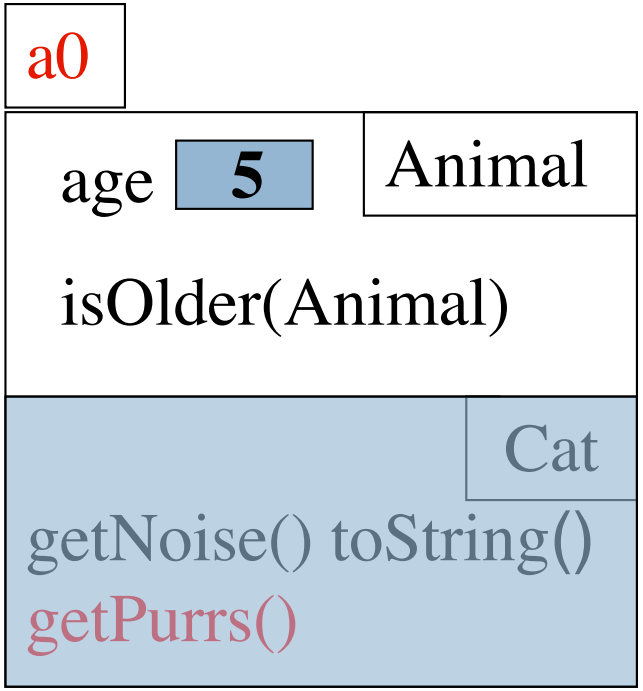
From v[k], see only what is in partition Animal and partitions above it.

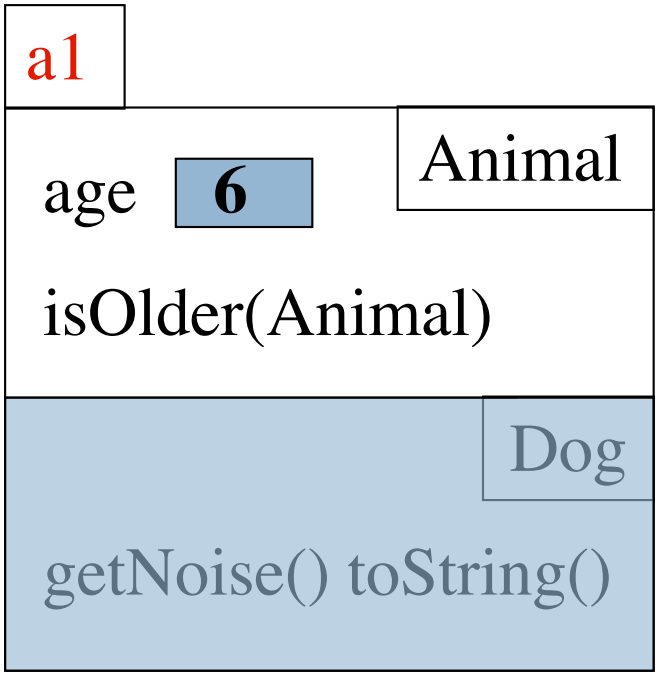
getPurrs() not in class Animal or Object. Calls are illegal, program does not compile:

v[0].getPurrs() v[k].getPurrs()

Components are in lower partitions, but can't see them







Casting objects

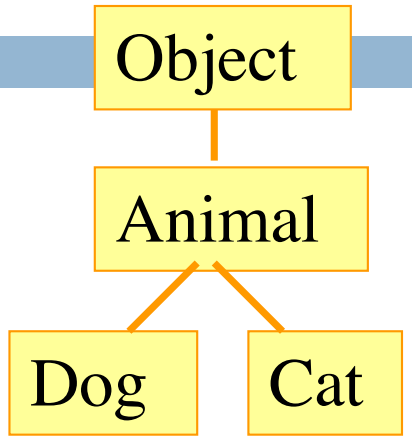
You know about casts like:

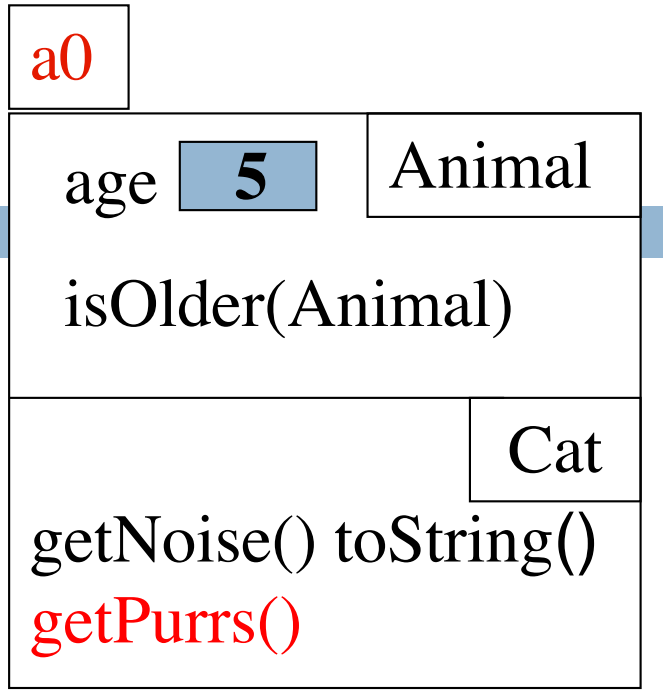
```
(int) (5.0 / 7.5)
(double) 6
double d= 5;     // automatic cast
```

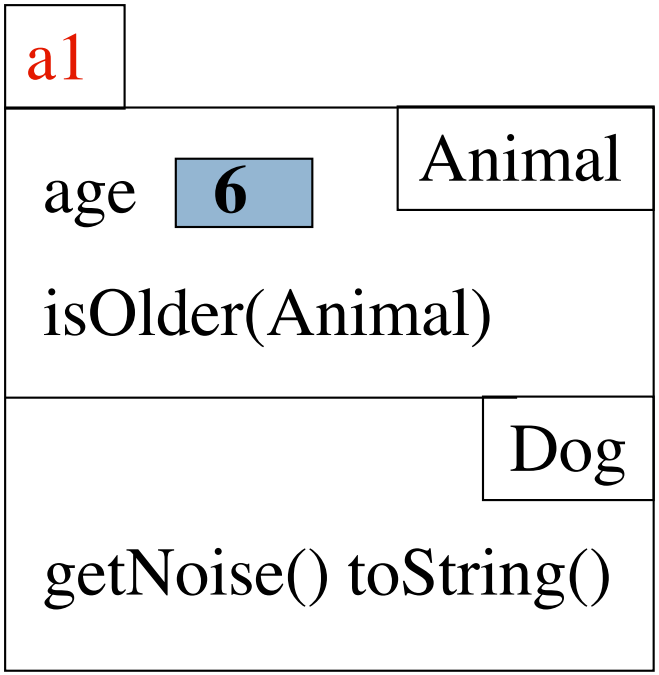
You can also use casts with class types:

```
Animal h= new Cat("N", 5);
Cat c= (Cat) h;
```

A class cast doesn't change the object. It just changes the perspective: how it is viewed!







Explicit casts: unary prefix operators

Rule: At run time, an object can be cast to the name of any partition that occurs within it —and to nothing else.

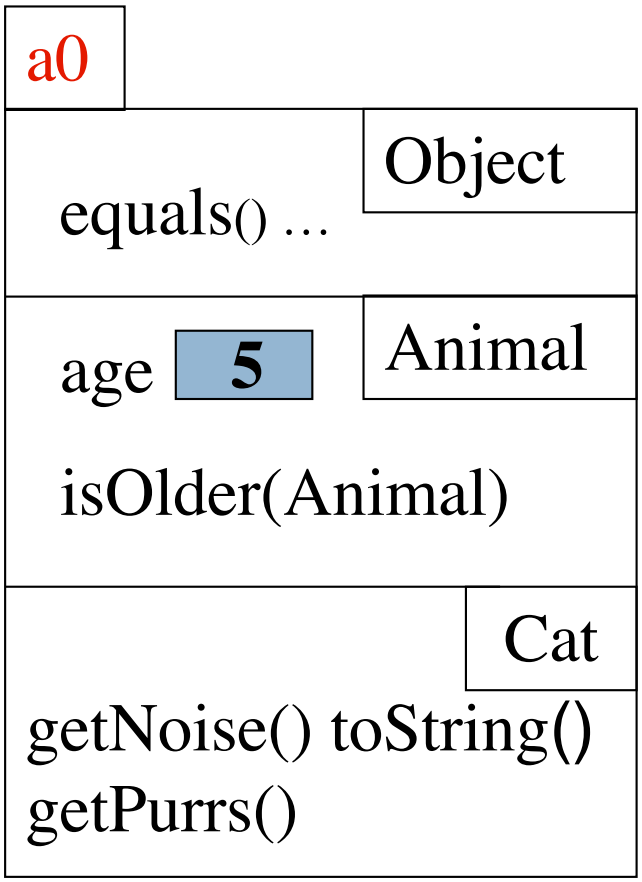
a0 can be cast to Object, Animal, Cat.

An attempt to cast it to anything else causes an exception

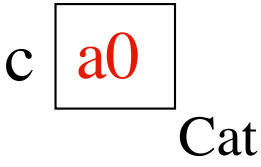
(Cat) c

(Object) c

(Animal) (Animal) (Cat) (Object) c



These casts don't take any time. The object does not change. It's a change of perception.



Implicit upward cast

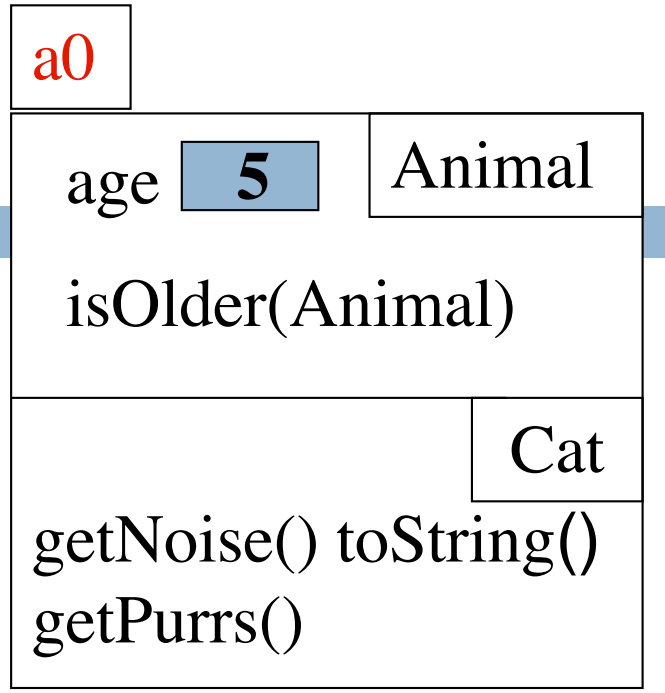

```
public class Animal {
    /** = "this Animal is older than h" */
    public boolean isOlder(Animal h) {
        return age > h.age;
    }
```

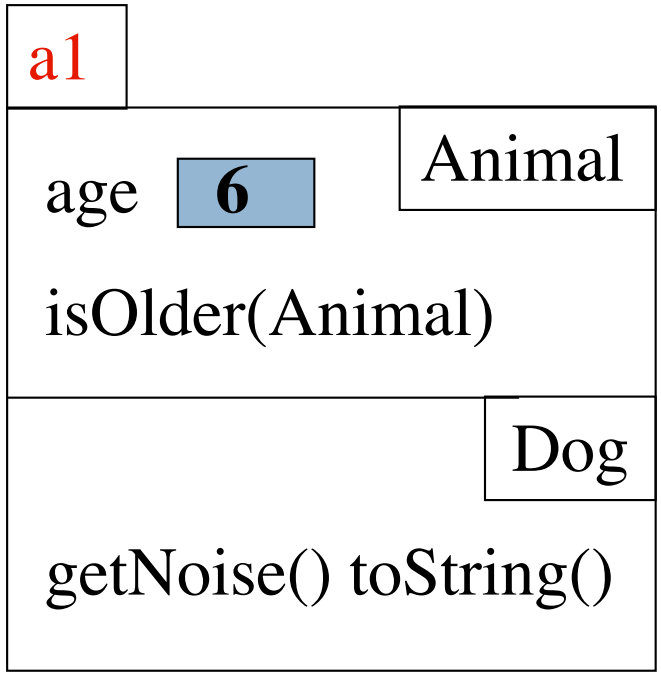
Call c.isOlder(d)

Variable h is created. a1 is cast up to class Animal and stored in h

Upward casts done automatically when needed

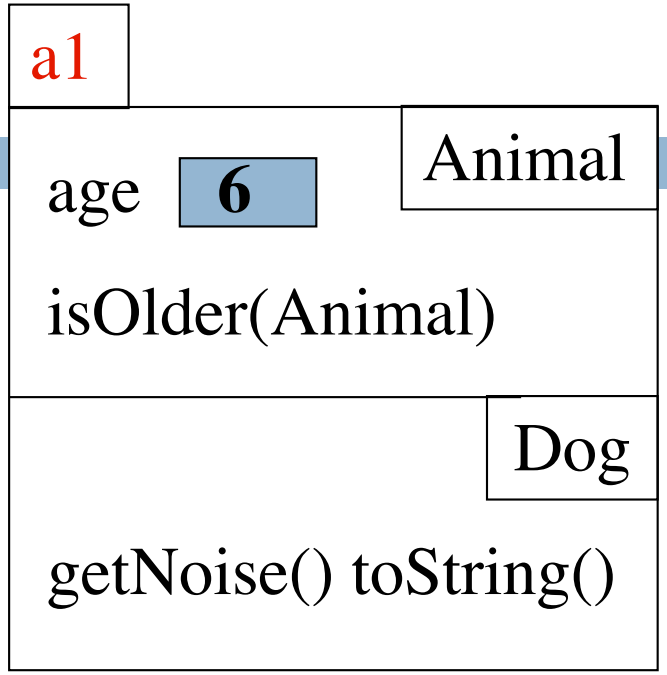
h a1 Animal c a0 Cat d a1 Dog





Example


```
public class Animal {
    /** = "this is older than h" */
    public boolean isOlder(Animal h) {
       return age > h.age;
    }
```



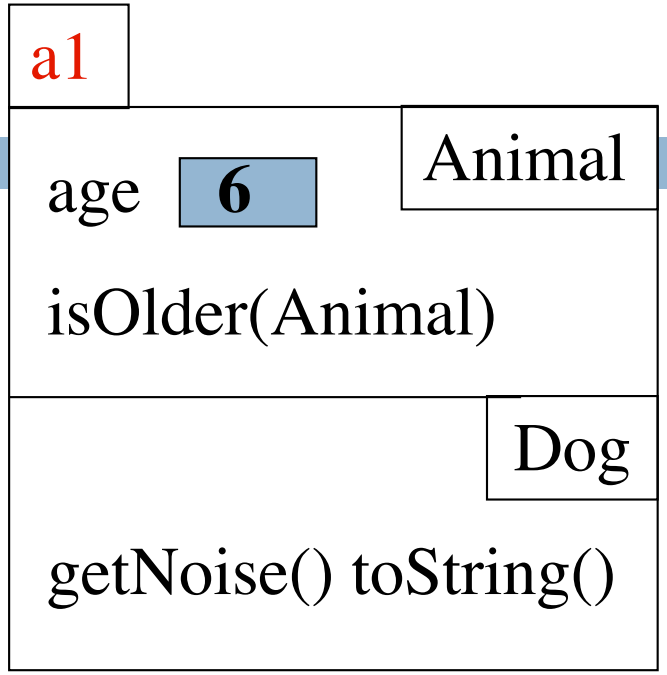
Type of h is Animal. Syntactic property.

Determines at compile-time what components can be used: those available in Animal

If a method call is legal, the overriding rule determines which implementation is called

Components used from h


```
public class Animal {
    /** = "this is older than h" */
    public boolean isOlder(Animal h) {
       return age > h.age;
    }
```



h.toString() OK —it's in class Object partition

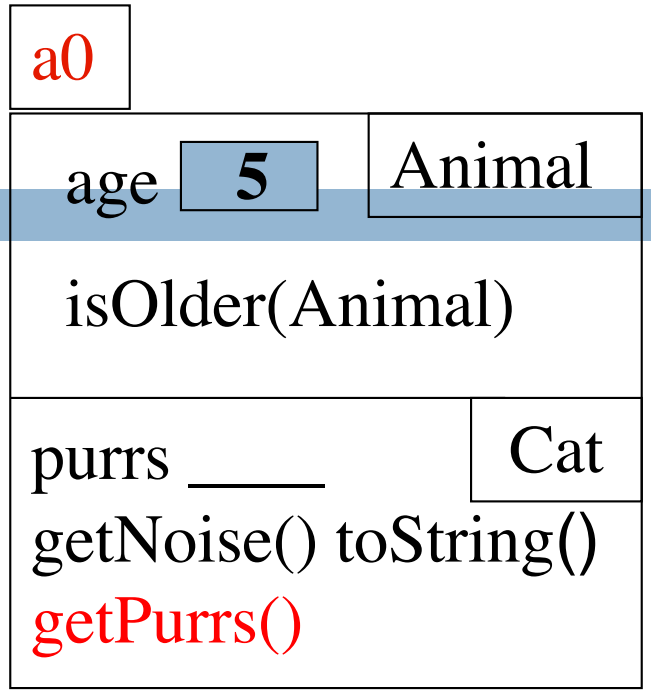
h.isOlder(…) OK —it's in Animal partition

h.getPurrs() ILLEGAL —not in Animal partition or Object partition

By overriding rule, calls toString() in Dog partition

Explicit downward cast


```
public class Cat extends Animal {
   private int purrs;
   /** return true iff ob is a Cat and its
     * fields have same values as this */
   public boolean equals(Object ob) {
                ?
     // { h is a Cat }
     if ( ! super.equals(ob) ) return false;
     Cat c= (Cat) ob ;  // downward cast
     return purrs == c.getPurrs();
}
```



h a0

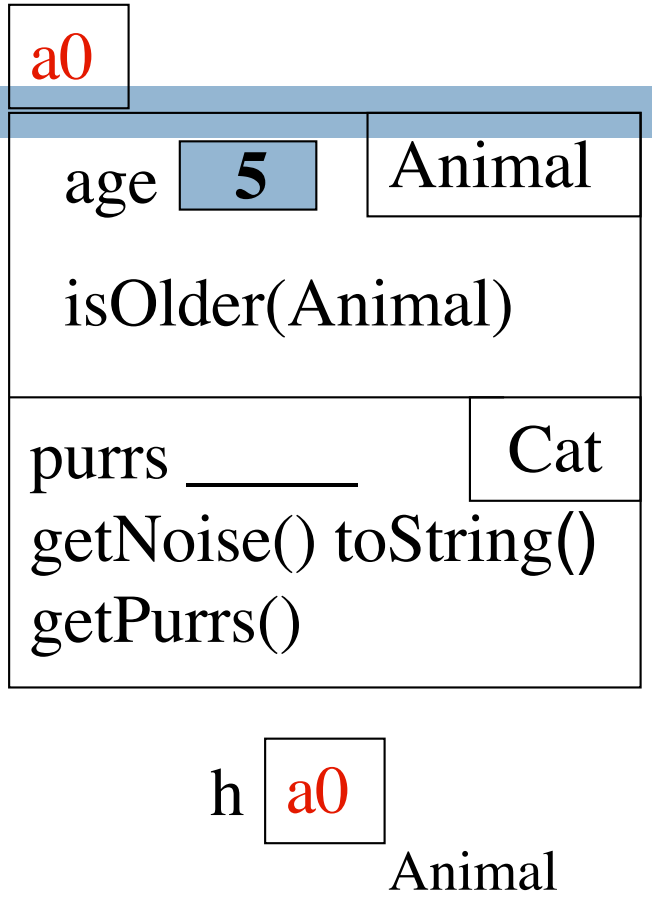
Animal

(Dog) ob leads to runtime error.

Don't try to cast an object to something that it is not!

Method getClass, explicit down cast


```
public class Cat extends Animal {
   private int purrs;
   /** return true iff ob is a Cat and its
     * fields have same values as this */
   public boolean equals(Object ob) {
     if ( ob.getClass() != getClass() )
           return false;
     // { h is a Cat }
     if ( ! super.equals(ob) ) return false;
     Cat c= (Cat) ob ;  // downward cast
     return purrs == c.getPurrs();
}
```

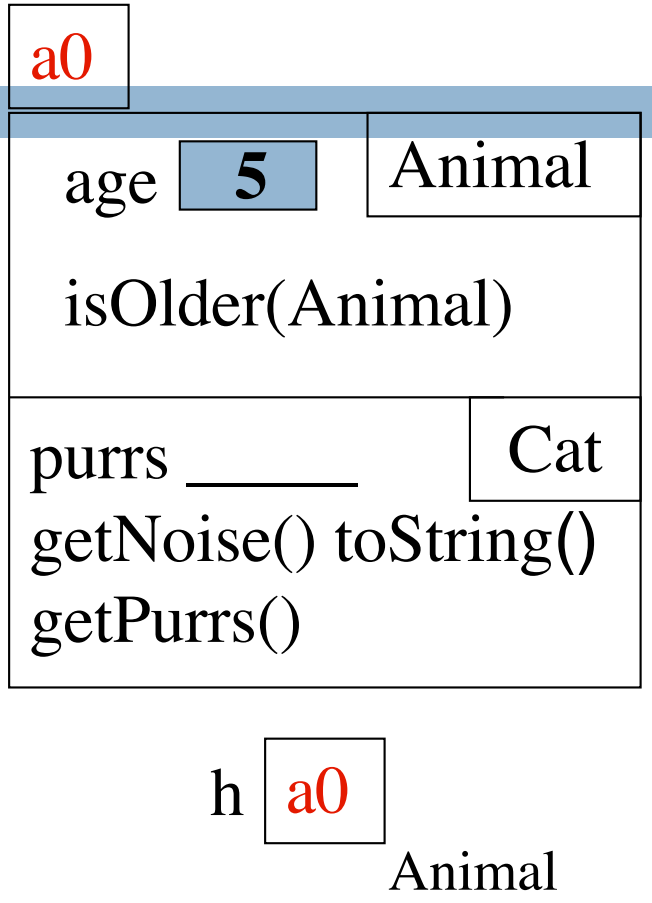


<object>.getClass() == <class-name>.class

true iff <object>'s bottom partition is <class-name>

A complete implementation of equals


```
public class Cat extends Animal {
   private int purrs;
   /** return true iff ob is a Cat and its
     * fields have same values as this */
   public boolean equals(Object ob) {
     if ( ob == null ||
          ob.getClass() != getClass() )
           return false;
     // { h is a Cat }
     if ( ! super.equals(ob) ) return false;
     Cat c= (Cat) ob ;  // downward cast
     return purrs == c.getPurrs();
}
```

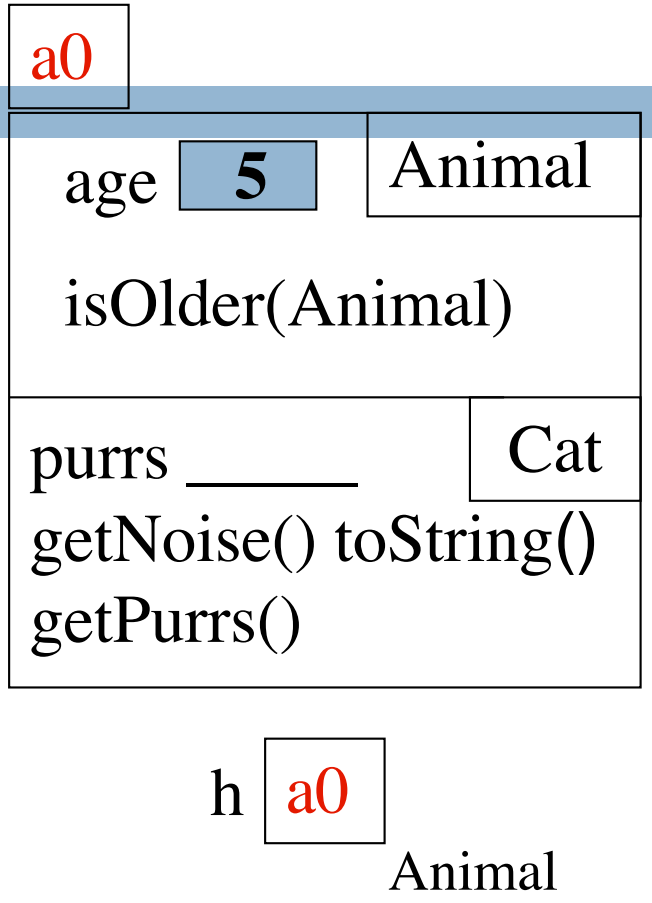


Check whether ob is null before calling getClass.

Operator instanceof


```
// Both are true.
if ( a0 instanceof Cat ) …
if ( a0 instanceof Animal ) …

// Only the first is true.
if ( a0.getClass() == Cat.class ) …
if ( a0.getClass() == Animal.class ) …
```



<object> **instanceof** <class-name>

true iff <object> has a partition for <class-name>

Opinions about casting

- Use of instanceof and downcasts can indicate bad design

DON'T:

```
if (x instanceof C1)
    do thing with (C1) x
else if (x instanceof C2)
    do thing with (C2) x
else if (x instanceof C3)
    do thing with (C3) x
```

DO:

```
x.do()
```

… where do is overridden in the classes C1, C2, C3

- But how do I implement equals() ?

That requires casting!