CS/ENGRD 2110
FALL 2018

Lecture 6: Consequence of type, casting; function equals
http://courses.cs.cornell.edu/cs2110

Overview references in *JavaHyperText*

2

- Quick look at arrays: array
- Casting among classes cast, object-casting rule
- 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
```

A2 is due Sunday

3

Everyone should get 100/100 since we gave you all the test cases you need.

Please look at the pinned Piazza note "Assignment A2" for information that is not in the handout and answers to questions.

Before Next Lecture…

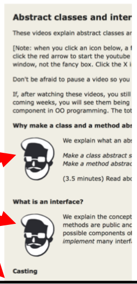
4

Follow the tutorial on **abstract classes and interfaces**, and watch <13 minutes of videos.

JavaHyperText
Abstract classes and interfaces

This will prepare you for Thursday's lecture.

Click these



Abstract classes and inter

These videos explain abstract classes ar

[Note: when you click an icon below, a f
click the red arrow to start the youtube
window, not the fancy box. Click the X i

Don't be afraid to pause a video so you

If, after watching these videos, you still
coming weeks, you will see them being
component in OO programming. The tot

Why make a class and a method abs

We explain what an abs

Make a class abstract s
Make a method abstrac

(3.5 minutes) Read abo

What is an interface?

We explain the concept
methods are public anc
possible components of
implement many interf.

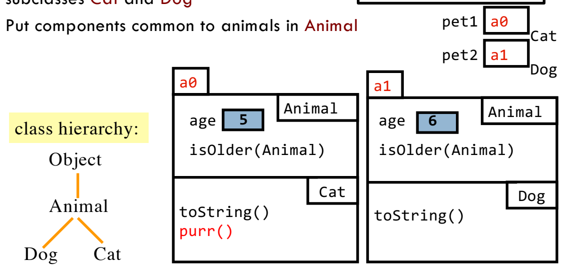
Casting

Classes we work with today

5

class Animal
subclasses Cat and Dog
Put components common to animals in Animal

```
Cat pet1= new Cat(5);
Dog pet2= new Dog(6);
```



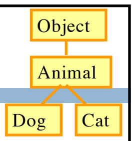
(Object partition is there but not shown)

6

Casting

Casting objects

7



You know about casts like:

(int) (5.0 / 7.5)

(double) 6

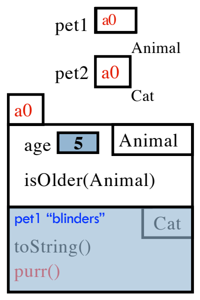
double d= 5;   // automatic cast

You can also use casts with class types:

Animal pet1= **new** Cat(5);

Cat pet2= (Cat) pet1;

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



Explicit casts: unary prefix operators

8

Object-casting rule: At runtime, 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.

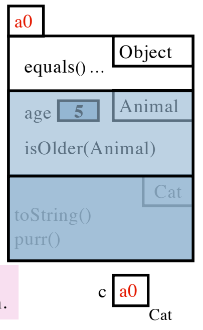
An attempt to cast it to anything else causes an exception

(Cat) c

(Object) c

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

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



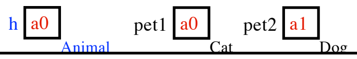
Implicit upward cast

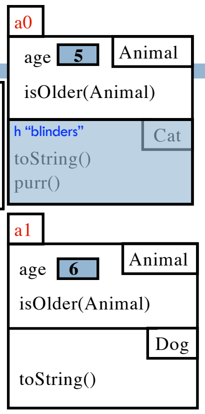
9

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

```
Cat pet1= new Cat(5);
Dog pet2= new Dog(6);
if (pet2.isOlder(pet1)) {...}
```

// pet1 is cast **up** to class Animal and stored in h

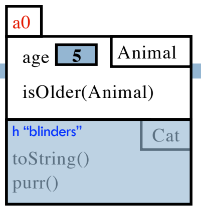




Components used from h

10

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

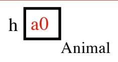


h.toString() OK —it's in class Object partition

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

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

Which toString() gets called? See slide 18.



11 Compile-time reference rule

Compile-time reference rule (v1)

see *JavaHyperText*

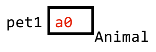
12

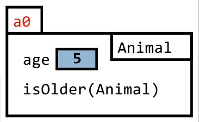
From a variable of type C, you can reference only methods/fields that are available in class C.

```
Animal pet1= new Animal(5);
pet1.purr();
```

obviously illegal

The compiler will give you an error.





Checking the legality of pet1.purr(...):

Since **pet1** is an Animal, purr must be declared in Animal or one of its superclasses.

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

Compile-time reference rule (v2)

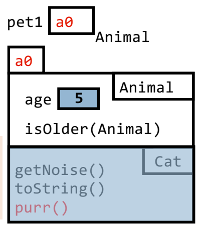
see *JavaHyperText*

From a variable of type C, you can reference only methods/fields that are available in class C.

```
Animal pet1= new Cat(5);
pet1.purr();
```

still illegal!

The compiler still gives you an error.



Checking the legality of pet1.purr(…):

Since **pet1** is an Animal, purr must be declared in Animal or one of its superclasses.

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

Why would we ever do this?

- Why would a variable of type Animal ever not have just an Animal in it?
- This is one of the beautiful things about OO programming!

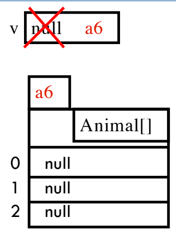
1. We want to use an Animal method (seen)
2. We want to keep a list of all our pets
   - Create an array of type Animal!

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

declaration of array v

Create array of 3 elements

Assign value of new-exp to v



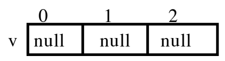
Assign and refer to elements as usual:

v[0]= **new** Animal(…);

…

a= v[0].getAge();

Sometimes use horizontal picture of an array:



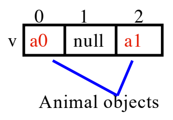
Consequences of a class type

```
Animal[] v;           // declaration of v
v= new Animal[3];     // initialization of v
v[0]= new Cat(5);     // initialization of 1st elem
v[2]= new Dog(6);
```

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.

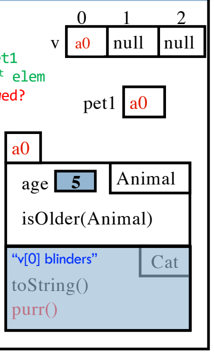


A variable's type:

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

Compile-time reference rule, revisited

```
Animal[] v;           // declaration of v
v= new Animal[3];     // initialization of v
Cat pet1= new Cat(5); // initialization of pet1
v[0]= pet1;           // initialization of 1st elem
v[0].purr();          // should this be allowed?
                      // will it compile?
```



Checking the legality of v[0].purr(…):

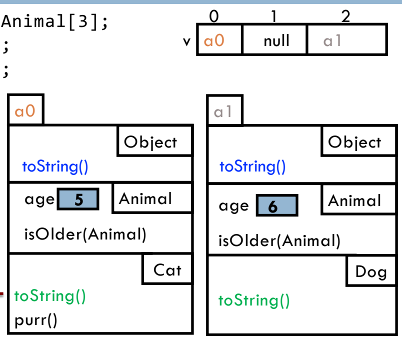
Since **v[0]** is an Animal, purr must be declared in Animal or one of its superclasses.

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

Bottom-up / Overriding rule revisited

```
Animal[] v= new Animal[3];
v[0]= new Cat(5);
v[2]= new Dog(6);
v[0].toString();
v[2].toString();
```

Which **toString()** gets called?



Bottom-up / Overriding rule says function toString in Cat partition

Equals

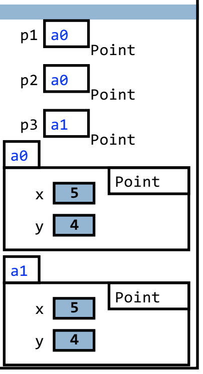
Example: Point Class

```
public class Point {
  public int x;
  public int y;

  public Point(int x, int y) {
    this.x= x;
    this.y= y;
  }
}
```

How **Object** defines equals(x)

```
public boolean equals(Object x) {
        return this == x;
}
```

```
Point p1= new Point(5,4);
Point p2= p1;

if (p1 == p2) {...}         // true?
if (p1.equals(p2)) {...}   // true?

Point p3= new Point(5,4);

if (p1 == p3) {...}         // true?
if (p1.equals(p3)) {...}   // true?
```



Using the `Point` class as defined in previous slide.

Can define equals for your own class!

Can I define it any way I like?

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

Java spec says:

- Reflexive
- Symmetric
- Transitive

(click on the link to see what these are)

How do we define equality for a Point?

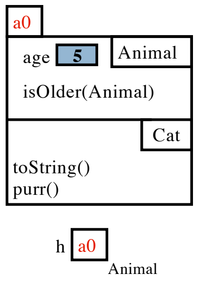
```
/** return “obj is a Point and
    obj and this have the same x and y fields” */
@Override
public boolean equals(Object obj) { // why Object?
      // how can we access the x y fields
      // if this is an Object?


}
```

Use operator **instanceof**

ob **instanceof** C

true iff ob has a partition named C

h **instanceof** Object	true
h **instanceof** Animal	true
h **instanceof** Cat	true
h **instanceof** JFrame	false



How do we define equality for a Point?

25

```
/** return "obj is a Point and
     obj and this have the same x and y fields"  */
@Override
 public boolean equals(Object obj) {
   if (!(obj instanceof Point))
       return false;
   Point p= (Point)obj;
   return (x == p.x && y == p.y);
 }
```

Opinions about casting

26

Use of instanceof and down-casts can indicate bad design

DON'T:	DO:

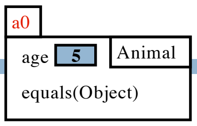
if (x instanceof C1)
  do thing with (C1) x
else if (x instanceof C2)
  do thing with (C2) x
else if (x instanceof C3)
  do thing with (C3) x
	x.do()
	
	… where do is overridden in the classes C1, C2, C3

But how do I implement equals() ?

That **requires** casting!

Equals in Animal

27



```
public class Animal {
   private int age;
   /** return true iff this and obj are of the same class
     * and their age fields have same values */
   public boolean equals(Object obj) {


      // how to check that objects are of the
      // same class??

}
```

Use function **getClass**

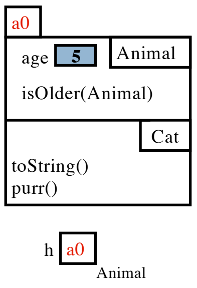
28

h.getClass()

Let Cat be the lowest partition of object h

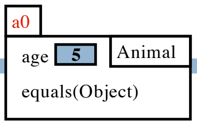
Then h.getClass() == Cat.class

h.getClass() != Animal.class



Equals in Animal

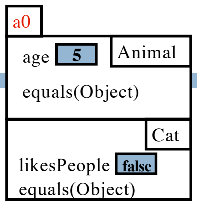
29



```
public class Animal {
   private int age;
   /** return true iff this and obj are of the same class
     * and their age fields have same values */
   public boolean equals(Object obj) {

      if (obj == null || getClass() != obj.getClass())
          return false;

      Animal an= (Animal) obj;  // cast obj to Animal!!!!

      return age == an.age;  // downcast needed to reference age
}
```

Equals in Cat

30



```
public class Animal {
   private int age;
   /** return true iff this and ob are of
     * same class and their age fields
     * have same values */
   public boolean equals(Object ob) {...}
```

```
public class Cat extends Animal {
   private boolean likesPeople;
   /** return true iff this and ob are of same class
     * and age and likesPeople fields have same values*/
   public boolean equals(Object obj) {
         if (!super.equals(obj)) return false;

         Cat c1= (Cat) obj;   // downcast is necessary!

         return likesPeople == c1.likesPeople;
   }
}
```