Reminder: A1 due tonight
Today’s topics

- Casting, object-casting rule
- Compile-time reference rule
- Quick look at arrays
- Implementing equals, method getClass

- Review on your own if you need to: while and for loop
Classes we work with today

class Animal
subclasses Cat and Dog

Put components common to animals in Animal

class hierarchy:
Object
   Animal
      Dog
      Cat

(Object partition is there but not shown)
Casting
Casting objects

You know about casts like:

(\texttt{int}) (5.0 / 7.5)

(\texttt{double}) 6

double d = 5;    // cast implicit

You can also use casts with class types:

Animal pet1 = new Cat(5);    // cast implicit

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

**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 a `ClassCastException`.

(Cat) `c`

(Object) `c`

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

The **object** does not change.

The **perception** of it changes.
Implicit upward cast

```java
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
Compile-time reference rule
Compile-time reference rule (v1)

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

```
Animal pet1 = new Animal(5);
int m = pet1.purrs();
```

The compiler will give you an error.

Checking the legality of \( \text{pet1.purrs(…)} \):
Since \( \text{pet1} \) is an Animal, \( \text{purrs} \) is legal only if it is declared in Animal or one of its superclasses.

From an Animal variable, can use only methods available in class Animal.
Quiz: Which references are legal?

A. `h.toString()`
   OK — it’s in class `Object` partition

B. `h.isOlder(...)`
   OK — it’s in `Animal` partition

C. `h.purrs()`
   ILLEGAL — not in `Animal` partition or `Object` partition
Arrays
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:
Array elements may be subclass objects

Animal[] v; // declaration of v
v= new Animal[3]; // initialization of v
v[0]= new Cat(5); // initialization of 1\textsuperscript{st} elem
v[2]= new Dog(6); // initialization of 2\textsuperscript{nd} elem

The type of \texttt{v} is \texttt{Animal[]}
The type of each \texttt{v[k]} is \texttt{Animal}
Compile-time reference rule (CTRR), applied

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
int m = v[0].purrs();  // is this allowed?

Not allowed!
Type of v[0] is Animal.
CTRR: May reference only methods available in Animal.
purrs is not declared in Animal or one of its superclasses.
Animal[] v = new Animal[3];
v[0] = new Cat(5);
v[2] = new Dog(6);
v[0].toString();

Which `toString()` gets called?

Bottom-up / Overriding rule says function `toString` in Cat partition
Equals
public class Point {
    public int x;
    public int y;

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

Object.equals

toString()

equals(Object o)

Point@01fb

x

y
How **Object** defines `equals(o)`

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

```java
Point p1 = new Point(5, 4);
Point p2 = p1;
if (p1 == p2) {...} // true?
if (p1.equals(p2)) {...} // true?

Point p3 = new Point(5, 4);
Point p4 = new Point(5, 4);
Point p5 = new Point(5, 4);
Point p6 = new Point(5, 4);
```

- `p1` and `p2`: `Point` objects with coordinates (5, 4)
- `p3` and `p4`: `Point` objects with coordinates (5, 4)
- `p5` and `p6`: `Point` objects with coordinates (5, 4)
Defining equality for your own class

- **Principle:** two objects should be equal iff their behavior is indistinguishable

- **Specification:** Object.equals has a specification you must obey: reflexive, symmetric, transitive
  
  https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

- Reflexive \( x . \text{equals}(x) \)
- Symmetric \( x . \text{equals}(y) \) iff \( y . \text{equals}(x) \)
- Transitive if \( x . \text{equals}(y) \) and \( y . \text{equals}(z) \) then \( x . \text{equals}(z) \)

- (and more...)
Are any of these equal?

No! All have distinguishable behavior.
public class Point {
    /** return “this and obj are of the same class, and this and obj have the same x and y fields” */
    @Override
    public boolean equals(Object obj) {
        How can we tell whether this and obj are of the same class?
    }
}
Method `getClass()` returns the class of the lowest partition in the object.

Point class is an object of class `Class` representing class `Point`.

`p.getClass() == Point.class`
Function `getClass` and static field `class`

- `h.getClass() == Cat.class`
- `h.getClass() != Animal.class`
- `h.getClass() != Object.class`

Method `getClass()` returns the class of the lowest partition in the object.
Equality for Points

/** return “this and obj are of the same class and this and obj have the same x and y fields” */

@Override

public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass())
        return false;
    Point p = (Point) obj; // downcast to reference Point fields

    return x == p.x && y == p.y;
}


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;
        return age == an.age;
    }
}
Casting advice

function equals() requires casting

But, use of explicit down-casts can indicate bad design

DON’T:

if ( … )
    do something with (C1) x
else if ( … )
    do something with (C2) x
else if (…)  
    do something with (C3) x

DO:

x.do()

… where do() is overridden in classes C1, C2, C3