Overview ref in text and JavaSummary.pptx

- Quick look at arrays slide 50-55
- Casting among classes C.33-C.36 (not good) slide 34-41
- Static/Dynamic types (apparent/real types) slide 34-41
- Operator instanceof slide 40
- Function equals slide 37-41

Homework. Learn about while/ for loops in Java. Look in text.

```java
while ( <bool expr> ) { … }              // syntax

for (int k= 0; k < 200; k= k+1) { … }   // example
Classes we work with today

Work with a class `Animal` and subclasses like `Cat` and `Dog`

Put components common to animals in `Animal Object`, partition is there but not shown

```
Object
  Animal
    Dog
    Cat
```

```
class hierarchy:
```

```
Animal
  age
  Animal(String, int)
  isOlder(Animal)
  Cat(String, int)
  getNoise() toString()
  Cat
    getWeight()

Dog(String, int)
  getNoise() toString()

Cat(String, int)
```
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:**

```
    0  1  2
    v  null  null  null
```
Which function is called by \( v[0].toString() \)?

Remember, `partition` Object contains `toString()`
Each element $v[k]$ is of type `Animal`. Its declared type:
- static type — known at compile-time
- apparent type

Should this call be allowed? Should program compile?

```java
v[0].getWeight()
```
Each element $v[k]$ is of (static) type `Animal`.
From $v[k]$, see only what is in partition `Animal` and partitions above it.

Components are in lower partitions, but can’t see them.

- $v[0]$ = `Cat(String, int)`
- $v[1] = null$
- $v[2] = 'a1'$

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

$$v[0].\text{getWeight()} \ v[k].\text{getWeight()}$$
Casting up class hierarchy

You know about casts like

(int) (5.0 / 7.5)

(double) 6

double d = 5;  // automatic cast

We now discuss casts up and down the class hierarchy.

Animal h = new Cat(“N”, 5);
Cat c = (Cat) h;
Implicit upward cast

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

Call `c.isOlder(d)`

- `h` is created.
- `a1` is cast up to class `Animal` and stored in `h`.
- Upward casts done automatically when needed.

```
Animal a0
age 5
Animal
Animal(String, int)
Cat(isOlder(Animal))
Cat
getNoise() toString()
getWeight()

Animal a1
age 6
Animal
Animal(String, int)
Dog(isOlder(Animal))
Dog
getNoise() toString()
```
**Explicit casts: unary prefix operators**

**Principle:** you may cast an object to the name of any partition that occurs within it—and to nothing else.

- `a0` maybe 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`

<table>
<thead>
<tr>
<th><code>a0</code></th>
<th><code>equals()</code> ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Object</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>age 5</code></th>
<th><code>Animal</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal(String, int)</td>
<td></td>
</tr>
<tr>
<td><code>isOlder(Animal)</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>Cat(String, int)</code></th>
<th><code>Cat</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getNoise()</code> <code>toString()</code> <code>getWeight()</code></td>
<td></td>
</tr>
</tbody>
</table>

These casts don’t take any time. The object does not change. It’s a change of perception.
Static/dynamic types

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

**Static or apparent** type of h is Animal. Syntactic property
Determines at compile-time what components can be used: those available in Animal

**Dynamic or real** type of h is Dog. Semantic/runtime property
If a method call is legal, dynamic type determines which one is called (overriding one)
Components used from h

```java
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.getWeight() ILLEGAL — not in Animal partition or Object partition

By overriding rule, calls toString() in Cat partition
Explicit downward cast

```java
public class Animal {
    // If Animal is a Cat, return its weight; otherwise, return 0.
    public int checkWeight(Animal h) {
        if (! /* condition */) {
            return 0;
        } // { h is a Cat }
        Cat c = (Cat) h; // downward cast
        return c.getWeight();
    }
}
```

(Dog) h leads to runtime error.
Don’t try to cast an object to something that it is not!
public class Animal {
    // If Animal is a cat, return its weight; otherwise, return 0.
    public int checkWeight(Animal h) {
        if (! (h instanceof Cat))
            return 0;
        // { h is a Cat }
        Cat c = (Cat) h; // downward cast
        return c.getWeight();
    }
}

h a0 Animal

Cat(String, int) Cat
getNoise() toString() getWeight()

<object> instanceof <class>
true iff object is an instance of the class — if object has a partition for class
public class Object {
    /** Return true iff this object is the same as ob */
    public boolean equals(Object b) {
        return this == b;
    }
}

x.equals(y) is same as x == y except when x is null!

This gives a null-pointer exception: null.equals(y)
Override function equals in a class to give meaning to:

“these two (possibly different) objects of the class have the same values in some of their fields”

For those who are mathematically inclined, like any equality function, equals should be reflexive, symmetric, and transitive.

Reflexive: b.equals(b)
Symmetric: b.equals(c) = c.equals(b)
Transitive: if b.equals(c) and c.equals(d), then b.equals(d)
Function equals in class Animal

```java
public class Animal {
    /** = “h is an Animal with the same values in its fields as this Animal” */
    public boolean equals(Object h) {
        if (!(h instanceof Animal))
            return false;
        Animal ob = (Animal) h;
        return name.equals(ob.name) && age == ob.age;
    }
}
```

1. Because of \textit{h is an Animal} in spec, need the test \textit{h instanceof Animal}
public class Animal {
    /** = “h is an Animal with the same values in its fields as this Animal” */
    public boolean equals (Object h) {
        if (!(h instanceof Animal))
            return false;
        Animal ob = (Animal) h;
        return name.equals(ob.name) && age == ob.age;
    }
}

2. In order to be able to reference fields in partition Animal, need to cast h to Animal
3. Use `String equals` function to check for equality of `String` values. Use `==` for primitive types.
Why can’t the parameter type be Animal?

public class Animal {
    /** = “h is an Animal with the same values in its fields as this Animal” */
    public boolean equals(Animal h) {
        if (!(h instanceof Animal))
            return false;
        Animal ob = (Animal) h;
        return name.equals(ob.name) && age == ob.age;
    }
}

What is wrong with this?
Recitation this week: VERY important

Recitation this week is about

abstract classes
interfaces

Learn:

☑ Why we may want to make a class abstract
☑ Why we may want to make a method abstract
☑ An interface is like a very restricted abstract class, with different syntax for using it.

Don’t miss recitation