The fattest knight at King Arthur's round table was Sir Cumference. He acquired his size from too much pi.

Overview references in JavaHyperText
- 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 Thursday

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...

Follow the tutorial on abstract classes and interfaces, and watch less than 13 minutes of videos.

Visit JavaHyperText and click on Abstract classes and interfaces

This will make Thursday's lecture far more understandable.

Classes we work with today

Work with a class Animal and subclasses like Cat and Dog

Put components common to animals in Animal

Object

Animal

Dog

Cat

class hierarchy:

age

isOlder(Animal)

getNoise() toString()

getPurr()
Consequences of a class type

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

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

Compile-time reference rule: From a variable of type C, can reference only methods/fields that are available in class C.

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.]

Compile-time reference rule: From a variable of type C, can reference only methods/fields that are available in class C.

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

Compile-time reference rule: From a variable of type C, can reference only methods/fields that are available in class C.

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.]

Compile-time reference rule: From a variable of type C, can reference only methods/fields that are available in class C.

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>a0</td>
<td>null</td>
<td>a1</td>
</tr>
</tbody>
</table>

Should this call be allowed? Should program compile?

v[0].getPurrs()

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>Animal</td>
<td>age 5</td>
<td>Animal</td>
</tr>
<tr>
<td>a1</td>
<td>Animal</td>
<td>age 6</td>
<td>Animal</td>
</tr>
</tbody>
</table>

isOlder(Animal)

catNoise() toString()

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.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>Animal</td>
<td>age 5</td>
<td>Animal</td>
</tr>
<tr>
<td>a1</td>
<td>Animal</td>
<td>age 6</td>
<td>Animal</td>
</tr>
</tbody>
</table>

isOlder(Animal)

catNoise() toString()

getPurrs()

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>null</td>
<td>a1</td>
<td></td>
</tr>
</tbody>
</table>

Animal

getPurrs()

Should this call be allowed? Should program compile?

v[k].getPurrs()

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>Animal</td>
<td>age 5</td>
<td>Animal</td>
</tr>
<tr>
<td>a1</td>
<td>Animal</td>
<td>age 6</td>
<td>Animal</td>
</tr>
</tbody>
</table>

isOlder(Animal)

catNoise() toString()

getPurrs()

Each element v[k] is of type Animal.

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>Animal</td>
<td>age 5</td>
<td>Animal</td>
</tr>
<tr>
<td>a1</td>
<td>Animal</td>
<td>age 6</td>
<td>Animal</td>
</tr>
</tbody>
</table>

isOlder(Animal)

catNoise() toString()

getPurrs()

Should this call be allowed? Should program compile?

v[k].getPurrs()

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>null</td>
<td>a1</td>
<td></td>
</tr>
</tbody>
</table>

Animal

getPurrs()

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>Animal</td>
<td>age 5</td>
<td>Animal</td>
</tr>
<tr>
<td>a1</td>
<td>Animal</td>
<td>age 6</td>
<td>Animal</td>
</tr>
</tbody>
</table>

isOlder(Animal)

catNoise() toString()

getPurrs()

Each element v[k] is of type Animal.

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>null</td>
<td>a1</td>
<td></td>
</tr>
</tbody>
</table>

Animal

getPurrs()

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

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
(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;
}

}

Function .equals()

Function .equals(ob) returns true if objects h and ob are equal, where equality depends on the class.

Here, we mean all corresponding fields are equal.

h.equals(): true
h.equals(k): true
h.equals(): false

Not Java

h.equals(a0): true
h.equals(a1): true
h.equals(a2): false

j.equals(a0): true
j.equals(a1): true
j.equals(a2): false

k.equals(a0): true
k.equals(a1): true
k.equals(a2): false

age 5
An
equals(Object)

age 5
An
equals(Object)

age 5
An
equals(Object)
Function `h.equals(ob)`

Function `h.equals(ob)` returns true if objects `h` and `ob` are equal, where equality depends on the class. Here, we mean all corresponding fields are equal.

```
Function h.equals(ob) returns true if objects h and ob are equal, where equality depends on the class. Here, we mean all corresponding fields are equal.
```

```
| a0 | age _5_ An equals(Object) |
| a1 | age _2_ An equals(Object) |
| a2 | age _3_ An equals(Object) |
|    | noise _"p"_ Cat equals(Object) |

This function checks equality of age
This function checks equality of noise
```

```
What is value of a1.equals(a0)?
Obviously, h.equals(ob) has to check that the classes of h and ob are the same
This function checks equality of age
This function (1) Calls super-class equality
(2) Checks equality of noise
```

```
Use function getClass

Let Cat be the lowest partition of object h
Then h.getClass == Cat.class
h.getClass != Animal.class
```

```
Equals in Animal

public class Animal {
  private int age;
  /** return true iff this and ob are of the same class *
   * and their age fields have same values */
  public boolean equals(Object ob) {
    if (ob == null || ob.getClass() != Class.getAn().getClass()) return false;
    Animal an = (Animal) ob; // cast ob to Animal!!!!!
    return age == an.age; // downcast was needed to reference age
  }
}
```
public class Cat extends 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) {
        if (!super.equals(ob)) return false;
        Cat ca = (Cat) ob; // downcast is necessary!
        return noise == ca.noise; // needed to reference noise
    }
}

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!

Equals in Cat

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) {}