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.

while (<bool expr>) { ... } // syntax
for (int k= 0; k < 200; k= k+1) { ... } // example
View of object from static type

![Diagram of object view](image)

- 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.
- \(v[0] = \text{null} \quad v[1] = a_1 \quad v[2] = a_1\) in Animal.

Casting up class hierarchy

![Diagram of class hierarchy](image)

- You know about casts like:
  - (int) \(5.0 / 7.5\)
  - (double) \(6\)
- \(double d = 5\); // automatic cast

Implicit upward cast

![Diagram of implicit cast](image)

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

- Call: \(\text{cisOlder(d)}\)
  - \(h\) is created. \(a_1\) is cast up to class Animal and stored in \(h\).
- Upward casts done automatically when needed.

Explicit casts: unary prefix operators

![Diagram of explicit casts](image)

- Principle: you may cast an object to the name of any partition that occurs within — and to nothing else.
- \(a_0\) maybe cast to Object, Animal, Cat.
- An attempt to cast it to anything else causes an exception.
- \((\text{Cat}) c\)
- \((\text{Object}) c\)
- \((\text{Animal}) (\text{Animal}) (\text{Cat}) (\text{Object}) c\)

Static/dynamic types

![Diagram of static/dynamic types](image)

- Static or apparent type of \(h\) is Animal. Syntactic property.
- 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 \(h\)

![Diagram of components used](image)

- By overriding rule, calls \(\text{toString()}\) in Cat partition.
- \(h\).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.
### Explicit downward cast

```java
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();
    }
}
```

This leads to a runtime error:

```
(Dog) h leads to runtime error.
```

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

### Operator instanceof, explicit downward cast

```java
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();
    }
}
```

### Function equals

```java
public class Object {
    /** Return true iff this object is the same as ob */
    public boolean equals(Object b) {
        return this == b;
    }
}
```

This gives a null-pointer exception:

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

### Overriding function equals

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 h is an Animal in spec, need the test h instanceof Animal

### 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;
    }
}
```

2. In order to be able to reference fields in partition Animal, need to cast h to Animal
### 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;
    }
}
```

3. Use `String equals` function to check for equality of `String` values. Use `==` for primitive types.

### Why can't the parameter type be Animal?

```java
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 and 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.