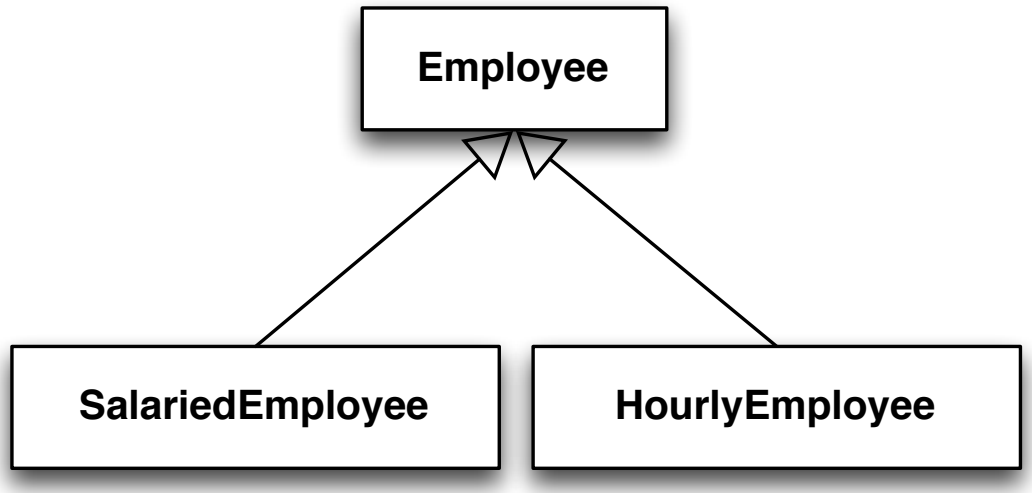
Inheritance, Part 3 of 3

Christopher Simpkins
chris.simpkins@gatech.edu

Our `Employee` Class Hierarchy

Recall our `Employee` class hierarchy:



Refactoring Common Code Into a Superclass

Let's move the definition of `disallowZeroesAndNegatives` into `Employee` so it will be shared (rather than duplicated) in `SalariedEmployee` and `HourlyEmployee`.

After cutting `disallowZeroesAndNegatives` from `SalariedEmployee` and `HourlyEmployee` and pasting it into `Employee`, `javac` tells us:

```
$ javac Employee.java HourlyEmployee.java SalariedEmployee.java
HourlyEmployee.java:25: cannot find symbol
symbol  : method disallowZeroesAndNegatives(double,double)
location: class HourlyEmployee
        disallowZeroesAndNegatives(anHourlyWage, aMonthlyHours);
        ^
SalariedEmployee.java:17: cannot find symbol
symbol  : method disallowZeroesAndNegatives(double)
location: class SalariedEmployee
        disallowZeroesAndNegatives(anAnnualSalary);
        ^
2 errors
```

Why did we get these errors?

`protected` Members

`private` members of a superclass are effectively not inherited by subclasses. To make a member accessible to subclasses, use `protected`:

```
public class Employee {
    protected void disallowZeroesAndNegatives(double ... args) {
        // ...
    }
    // ...
}
```

`protected` members

- are accessible to subclasses and other classes in the same package, and
- can be overriden in subclasses.

`protected` members provide encapsulation within a class hierarchy, `private` provides encapsulation within a single class.

Fitting Classes Into the Java Hierarchy

`java.lang.Object` defines several methods that are designed to be overriden in subclasses JLS §4.3.2:

- The method `equals` defines a notion of object equality, which is based on value, not reference, comparison.
- The method `hashCode` is very useful, together with the method `equals`, in hashtables such as `java.util.Hashmap`.
- The method `toString` returns a `String` representation of the object.
- The method `clone` is used to make a duplicate of an object
- The method `finalize` is run just before an object is destroyed.

A class hierarchy is also sometimes called a *framework*.

When to Override the `equals` Method

The default implementation of `equals` in `java.lang.Object` is object identity - each object `equals` only itself.
When should a class override `equals`?

- When logical equality differes from object identity, as is the case for *value* classes like `Date`
- When classes don't implement instance control.
  - Instance control means that a class ensures that there is only one instance of a class.
- When a suitable `equals` method is not provided by a superclass.

More important than recognizing *when* to override `equals` is knowing *how* to override `equals`.

How to Override the `equals` Method

Obey the general contract of `equals` (JLS), which says that `equals` defines an equivalence relation. So, for non-null references, `equals` is

- reflexive - any object `equals` itself
- symmetric - if `a.equals(b)` is true then `b.equals(a)` must be true
- transitive - if `a.equals(b)` and `b.equals(c)` are true then `a.equals(c)` must be true
- consistent - if `a` and `b` do not change between invocations of `a.equals(b)` or `b.equals(a)` then each invocation must return the same result
- a.equals(null) must always return `false`.

A Recipe for Implementing `equals`

Obeying the general contract of `equals` is easy if you follow these steps.

1. ensure that is not null
2. check this == that
3. check that instanceof this
4. cast that to this.class (guaranteed to work after instanceof test)
5. check equals on each “significant” field