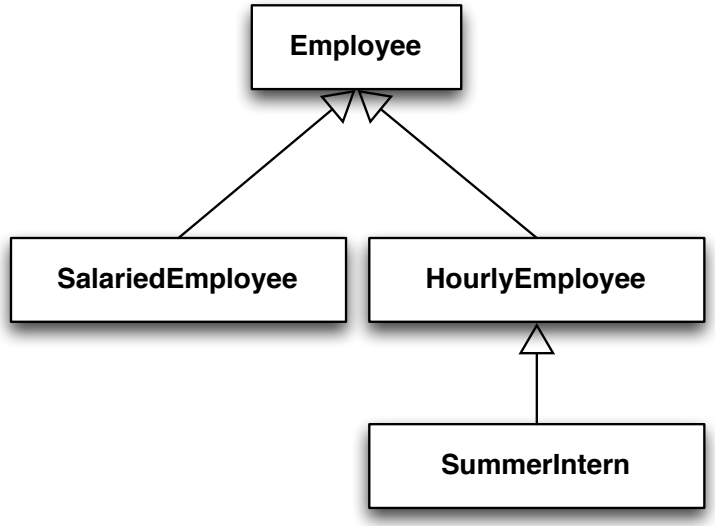
Introduction to Object-Oriented Programming

Object-Oriented Programming, Part 3 of 3

Christopher Simpkins
chris.simpkins@gatech.edu

The `Employee` Class Hierarchy

Let's add a summer intern class to our Employee hierarchy.



- We can get the payRoll for the current month by making use of the polymorphic `getMonthlyPay` method.
- What if we wanted to get the payroll for a particular month?

Let's overload `monthlyPay` so we can get the payroll for any month, not just the current month.

Enum Types

Enums are data types that have a predefined set of constant values (JLS §8.9, Java Enum Tutorial)
For example:

```
public enum Month {
    JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}
```

defines an enum type called `Month` that can take on only one of the predefined constants `Month.JAN`, `Month.FEB`, ..., `Month.DEC`

- Enum types are a class.
- Java automatically defines convenience methods for enum types, like `valueOf(String)` and `values()` (See the Enum API).
- Because they define a class, enum types can include programmer-defined additional constructors and methods.

Overloading Methods

An overloaded method is a set of methods with the same names but different signatures (parameter lists)[1] (JLS §8.4.9).

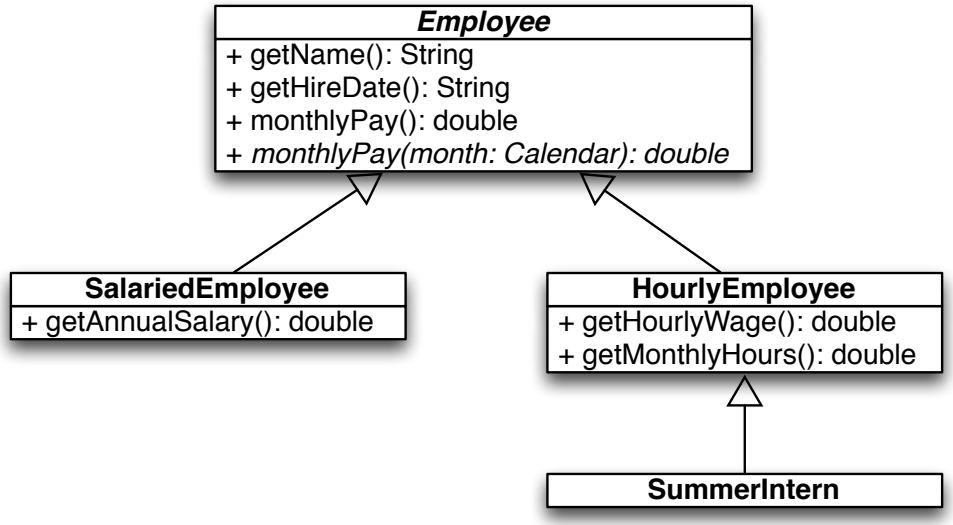
Here's an overloaded `monthlyPay` for `SummerIntern6`, along with a helper method demonstrating the use of the `Month` enum:

```
public double monthlyPay() {
    Date today = new Date();
    Month thisMonth = Month.values()[today.getMonth()];
    return monthlyPay(thisMonth);
}
public double monthlyPay(Month month) {
    return isSummer(month) ? super.monthlyPay() : 0.0;
}
private boolean isSummer(Month month) {
    return month == Month.JUN
        || month == Month.JUL
        || month == Month.AUG;
}
```

- In which classes should these methods be declared? Defined?

[1] More precisely, two methods with the same name whose signatures are not

The `Employee` Class Hierarchy in UML



- Italicized names are abstract (e.g., *Employee* is an abstract class, *+ getMonthlyPay(month: Month)* is an abstract method).
- We've only shown public methods (denoted by the '+' symbols in front of their names).
- Each class has all the public methods in its superclasses, and possibly additional methods.
- `SummerIntern6` only *specializes* `HourlyEmployee6`, that is, it modifies some behavior of its superclass but does not add any additional behavior.

Forecasting Payroll

Now with our overloaded `monthlyPay` method we can forecast payroll:

```
Company6 c = new Company6();
System.out.println("Monthly payroll this month: " +
  c.monthlyPayroll());
System.out.printf("Monthly payroll for May: %.2f%n",
                  c.monthlyPayroll(Month.MAY));
System.out.printf("Monthly payroll for June: %.2f%n",
                  c.monthlyPayroll(Month.JUN));
```

Inheritance Hinders Re-use

Recall the `disallowZeroesAndNegatives` method that we refactored so that it's in the `Employee` class and inherited by subclasses:

```
public abstract class Employee6 {
    protected void disallowZeroesAndNegatives(double ... args) {
        // ...
    }
}
```

- There's nothing about this method that is specific to `Employee`s
- `disallowZeroesAndNegatives` could be useful in other classes that are not part of the `Employee` class hierarchy.
- Since it's `protected`, it can't be used outside of the `Employee` class hierarchy or package.

In software engineering terms, we say that the code in `Employee` lacks *cohesion* - it has parts that aren't part of the *Employee* concept. Such a design hinders reuse.

Favor Composition over Inheritance

If we move these protected methods into a separate class, like ValidationUtils.java

```
public class ValidationUtils {

    public static void disallowNullArguments(Object ... args) { ... }

    public static void disallowZeroesAndNegatives(double ... args) {
     ... }
}
```

we can use them anywhere, e.g.,

```
    public Employee(String aName, Date aHireDate) {
        ValidationUtils.disallowNullArguments(aName, aHireDate);
        name = aName;
        hireDate = aHireDate;
    }
```

With this refactoring, we have our final versions of Employee.java, HourlyEmployee.java, and SalariedEmployee.java