Polymorphism

Introduction to Object-Oriented Programming

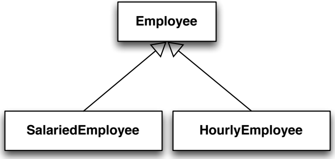
Today we'll learn how to combine all the elements of object-oriented programming in the design of a program that handles a company payroll. Object-oriented programming requires three features:

- Data abstraction with classes (encapsulation)
- Inheritance
- Dynamic method binding

That last part, dynamic method binding, provides for subtype polymorphism, which we'll learn today.

Class Hierarchies

UML class hierarchies depict the superclass-subclass relationships between families of related classes.



- `Employee` is the superclass of `HourlyEmployee` and `SalariedEmployee`
- `Employee` is more general than `HourlyEmployee` and `SalariedEmployee`, e.g., there at least as many `Employee` s as either `HourlyEmployee` s or `SalariedEmployee` s
- `HourlyEmployee` and `SalariedEmployee` are richer than `Employee` becuse they extend `Employee` with additional features

A SalariedEmployee Class

Let's add SalariedEmployee to our class hierarchy.

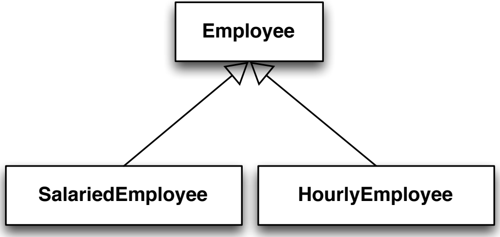
```
public final class SalariedEmployee3 extends Employee3 {

    private static final int MONTHS_PER_YEAR = 12;
    private final double annualSalary;

    public SalariedEmployee3(String aName, Date aHireDate,
                             double anAnnualSalary) {
        super(aName, aHireDate);
        Disallowzeroesandnegatives(anAnnualSalary);
        annualSalary = anAnnualSalary;
    }
    public double getAnnualSalary() {
        return annualSalary;
    }
    public double monthlyPay() {
        return annualSalary / MONTHS_PER_YEAR;
    }
    // ...
}
```

Our Employee Class Hierarchy

We now have all the classes in our hierarchy:



But our classes aren't well factored.

- SalariedEmployee3 and HourlyEmployee3 have duplicate copies of `disallowZeroesAndNegatives`
- SalariedEmployee3 and HourlyEmployee3 both have `monthlyPay` methods, but these methods are not polymorphic because they're not defined in Employee3

Let's refactor our `Employee` class hierarchy to give it a clean object-oriented design.

A Company Spec

Before we make `monthlyPay` polymorphic, we need an application to demonstrate why doing so is useful. Let's design a `Company` class with the following specs:

- A Company4 has exactly 9 employees (because we haven't learned about dynamically resized data structures yet)
- A company calculates its monthly payroll by adding up the monthly pay of each of its employees.
- A company can have any mix of hourly and salaried employees

That last bullet motivates the use of polymorphism.

Maintaining an Employee List

With our current class hierarchy, we need to maintain separate (partial) arrays of hourly and salaried employees. Because they're partial arrays we also need to keep track of how many of each type of employee we have.

```
public class Company {

    private HourlyEmployee[] hourlyEmployees;
    private int numHourlyEmployees = 10;
    private SalariedEmployee[] salariedEmployees;
    private int numSalariedEmployees = 10;

    public Company() {
        hourlyEmployees = new HourlyEmployee[numHourlyEmployees];
        salariedEmployees = new SalariedEmployee[numSalariedEmployees];
    }
}
```

Calculating Payroll the Hard Way

With our employee lists, calculating payroll is accomplished with two loops:

```
public class Company { // hypothetical

    public double monthlyPayroll() {
        double payroll = 0.0;
        for (int i = 0; i < numHourlyEmployees; ++i) {
            payroll += hourlyEmployees[i].monthlyPay();
        }
        for (int i = 0; i < numSalariedEmployees; ++i) {
            payroll += salariedEmployees[i].monthlyPay();
        }
        return payroll;
    }
    // ..
}
```

Seems reasonable. But ...

- What if we want to add a third type of employee?

Calculating Payroll the Easy Way

We'd like to be able to calculate payroll with a single loop over all employees:

```
public class Company4 {

    public double monthlyPayroll() {
        double payroll = 0.0;
        for (Employee employee: employees) {
            payroll += employee.monthlyPay();
        }
        return payroll;
    }
    // ..
}
```

Much cleaner and less error-prone (e.g., we don't have the book-keeping of two partial arrays). To be able to code like this we need to update the design of our `Employee` class hierarchy.

A More General Employee List

The first step is to store one array of `Employee` s:

```
public class Company4 {
    private Employee4[] employees;
    public Company4() {
        employees = ...;
    }
    public double monthlyPayroll() {
        double payroll = 0.0;
        for (int i = 0; i < employees.length; ++i) {
            payroll += employees[i].monthlyPay();
        }
        return payroll;
    }
}
```

Much better. But it doesn't compile. Why?

```
$ javac Company.java
Company.java:15: cannot find symbol
symbol : method monthlyPay()
location: class Employee
          payroll += employees[i].monthlyPay();
```

Abstract Classes

We need `Employee` to declare a `monthlyPay` method for subclasses to define. Since we don't have a general definition for `monthlyPay` suitable for `Employee`, `Employee` will need to be abstract.

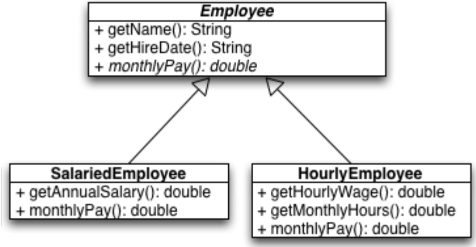
```
public abstract class Employee4 {
    // ...
    public abstract double monthlyPay();
}
```

An abstract class

- cannot be instantiated,
- may contain zero or more abstract methods, and
- subclasses must either provide an implementation for abstract methods, or be declared `abstract` themselves.

This makes sense for our `Employee4` class. We don't ever want to instantiate `Employee4` objects. `Employee4` simply defines the common aspects of all employees, with subclasses filling in the details.

The `Employee4` Class Hierarchy



- `Employee4` and its `monthlyPay` method are abstract.
- `monthlyPay` is polymorphic because it is overriden in subclasses.

Polymorphic Methods

```
public class Company4 {
    private Employee4[] employees;
    public double monthlyPayroll() {
        double payroll = 0.0;
        for (Employee4 employee: employees) {
            payroll += employees.monthlyPay();
        }
        return payroll;
    }
}
```

- The static type of the elements of `employees` is `Employee4`
- The dynamic type can be any subclass of `Employee4`, in this case they are all `SalariedEmployee4` and `HourlyEmployee4`
- When a method is invoked on an object, the method of the dynamic (run-time) type is used, no matter what the static (compile-time) type is.
- So though the static types of `employees` elements is `Employee`, the `monthlyPay` methods invoked on them are the ones defined in `SalariedEmployee4` and `HourlyEmployee4`.

Refactoring Duplicate Code in a Class Hierarchy

Recall the definition of `disallowZeroesAndNegatives`:

```
private void disallowZeroesAndNegatives(double ... args) {
    boolean shouldThrowException = false;
    String nonPositives = "";
    for (double arg: args) {
        if (arg <= 0.0) {
            shouldThrowException = true;
            nonPositives += arg + " ";
        }
    }
    if (shouldThrowException) {
        String msg = "Following arguments were <= 0: " + nonPositives;
        throw new IllegalArgumentException(msg);
    }
}
```

- This method is duplicated in `HourlyEmployee4` and `SalariedEmployee4`
- Let's move the definition of `disallowZeroesAndNegatives` into `Employee5` so it will be shared (rather than duplicated) in `SalariedEmployee5` and `HourlyEmployee5`.

`protected` Members

`private` members of a superclass are effectively invisible to subclasses.
To make a member accessible to subclasses, use `protected`:

```
public abstract class Employee5 {
    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 and package, `private` provides encapsulation within a single class.
Later we'll see a better way to re-use.

Implementation 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 `../code/employee/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

Closing Thoughts on Polymorphism

We've now seen two kinds of polymorphism:

- Ad-hoc polymorphism (method overloading), and
- Subtype polymorphism (overriding methods in subtypes).

Subtype polymorphism is core feature of OOP. Polymorphism makes it possible to reuse concepts in a way that makes programs extensible without requiring rewriting existing code - this is the open-closed principle.

In the next block we'll see one more kind of polymorphism: type parameter polymorphism, or parametric polymorphism.

Object-oriented Design

With encapsulation, inheritance, and polymorphism we have all the language features we need to employ three important object-oriented design principles:

- S ingle responsiblity principle: a module should only contain code related to the definition of the module
  - `Employee` classes contain only employee-related code, validation code is in utility class
- O pen-closed principle: open for extension, closed for modification
  - Can add new `Employee` subclasses without changing other classes in the `Employee` hierarchy or classes that use `Employee~s, such as ~Company`
- L liskov substitution principle: instances of subtypes should be substitutable wherever instances of supertypes are expected
  - A square is not a rectangle in an OO sense, but both are 2-D shapes

In CS 2340 you'll learn several more OO design principles and several patterns that employ them.