Introduction to Object-Oriented Programming
Polymorphism

Christopher Simpkins
chris.simpkins@gatech.edu

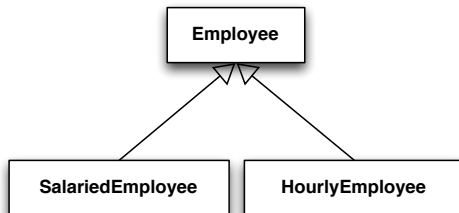# Introduction to Object-Oriented Programming

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

Class hierarchies depict the superclass-subclass relationships between families of related classes. Consider:

```
                    ┌──────────────┐
                    │   Employee   │
                    └──────────────┘
                          △ △
                      ╱        ╲
        ┌────────────────────┐  ┌────────────────────┐
        │  SalariedEmployee  │  │   HourlyEmployee    │
        └────────────────────┘  └────────────────────┘
```

- `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. Here are the important pieces:

```java
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:
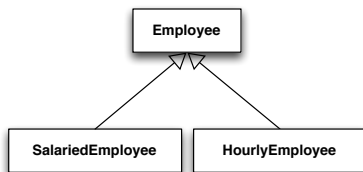
But our classes aren't well factored.

- `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 (becuase 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.

```java
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:

```java
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:

```java
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:

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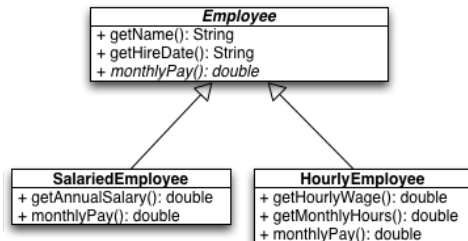
- `Employee4` and its `monthlyPay` method are abstract.
- `monthlyPay` is polymorphic because it is overriden in subclasses.

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`:

```java
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 overridden 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.

The `Employee` Class Hierarchy

Let's add a summer intern class to our Employee hierarchy.
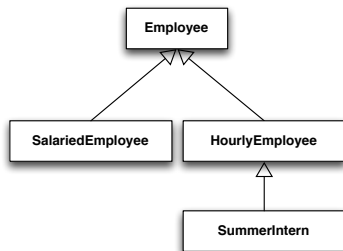
- We can get the payRoll for the current month by making use of the polymorphic `getMonthlyPay` method.
- What if we wanted to get the payroll for a particular month?

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:

```java
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.

Ad-Hoc Polymorphism: Overloaded 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:

```java
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
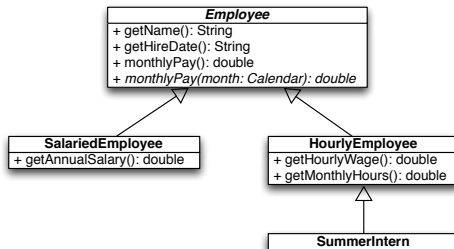
- Italicized names are abstract (e.g., *Employee* is an abstract class, + *getMonthlyPay(month: Month)* is an abstract method).
- We've only shown public methods (denoted by the '+' symbols in front of their names).
- Each class has all the public methods in its superclasses, and possibly additional 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 `montlyPay` 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:

```java
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

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 typically regarded as the core feature of OOP. Polymorphism makes it possible to reuse *concepts* in a way that makes programs extensible without requiring rewriting existing code.

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

Programming Exercise

Expand on the `Animal` and `Dog` exercise by making the following changes:

- Make the `speak` method in `Animal` abstract. What additional change to `Animal` will you have to make?
- Add a `Cat` class which overrides `speak` appropriately.
- Create a `Zoo` class that is just like `Kennel` except that it maintains an array of `Animal` (instead of `Dog`)