Introduction to Object-Oriented Programming

Inheritance, Part 1 of 2

Christopher Simpkins
chris.simpkins@gatech.edu

Programming in the Large

Software is complex. Three ways we deal with complexity:

- Abstraction - boiling a concept down to its essential elements, ignoring irrelevant details
- Decomposition - decompose system into packages, classes, functions
- Reuse - reuse library function in many diferent places

Today we introduce another kind of resuse: inheritance

What is inheritance?

 [1]

[1] Source: http://talentenbank.com/can-you-really-make-inheritance-into-a-good-financial-move-in-the-long-run

What is inheritance?

More like genetics ...

[2]

... but a programming concept that, like so much in CS, borrows a term from another field to leverage our intuition.

[2] Source: http://www.dnaftb.org/5/

Inheritance

Inheritance: deriving one class from another class.

```
public class Employee { ... }
public class HourlyEmployee extends Employee { ... }
public class SalariedEmployee extends Employee { ... }
```

- `Employee` is the *base class* or *superclass*
- `HourlyEmployee` and `SalariedEmployee` are *derived classes* or *subclasses*
- Subclasses *inherit* the interface and implementation of their superclass(es)
- `extends` is the Java syntax for inheriting from another class

Important idea to plant in your head now: subclassing is about concept reuse not merely implementation reuse. For example, `HourlyEmployee` *is-a* `Employee` conceptually.

Superclasses

Consider the superclass Employee1:[3]

```
public class Employee1 {
    private String name;
    private Date hireDate;

    public Employee1(String aName, Date aHireDate) {
        disallowNullArguments(aName, aHireDate);
        name = aName;
        hireDate = aHireDate;
    }
    public String getName() {
        return name;
    }
    public Date getHireDate() {
        return hireDate;
    } // and toString(), etc. ...
}
```

`Employee` defines the basic information needed to define any employee.

[3] Note that we'll number the versions of our Employee classes like we did with Card.

Subclasses

The `extends` clause names the direct superclass of the current class (JLS §8.1.4).
Here is a subclass of `Employee1`, HourlyEmployee1:

```
public class HourlyEmployee extends Employee {

    public HourlyEmployee(String aName, Date aHireDate) {
        super(aName, aHireDate);
    }
}
```

- `HourlyEmployee` inherits all the members of `Employee`
- `HourlyEmployee` can't access private members of `Employee` directly
- The `super` call in the constructor calls `Employee`'s constructor to initialize `HourlyEmployee` instances

The `HourlyEmployee` concept extends the `Employee` concept.

`super` Subtleties

- If present, an explicit `super` call must be the first statement in a constructor.
- If an explicit `super` call is not present and the superclass has a no-arg constructor, `super()` will implicitly be the first statement in any constructor
- If there is no no-arg constructor in a superclass (for example, if the superclass defines other constructors without explicitly defining a no-arg constructor), then subclass constructors must explicitly include a `super` call.

Together, these rules enforce an "inside-out" construction order for objects: the highest superclass piece of an object is initialzed first, followed by the second highest, and so on.

Subclass Constructors

Recall our definitions of `Employee1` and `HourlyEmployee1`.

```
public class Employee1 {
    // The only constructor in Employee
    public Employee1(String aName, Date aHireDate) {
        disallowNullArguments(aName, aHireDate);
        name = aName;
        hireDate = aHireDate;
    }
    // ...
}
```

```
public class HourlyEmployee1 extends Employee1 {

    public HourlyEmployee1(String aName, Date aHireDate) {
        super(aName, aHireDate);
    }
}
```

Would `HourlyEmployee1.java` compile if we left off the constructor definition?

Inherited Members

Given our previous definitions of `Employee1` and `HourlyEmployee1`, we can write code like this (from EmployeeDemo1):

```
DateFormat df = DateFormat.getDateInstance();
HourlyEmployee eva = new HourlyEmployee("Eva L. Uator",
                                        df.parse("February 18, 2013"));
System.out.println(eva.getName() + " was hired on "
                   + eva.getHireDate());
```

Note that

- we didn't have to define `getName` and `getHireDate` in `HourlyEmployee`
- our current implementation of `HourlyEmployee` doesn't add anything to `Employee`

Subclasses Specialize Superclasses

We define subclasses to *extend* or *specialize* the functionality of their superclasses. Let's add suitable extensions to `HourlyEmployee`:[4]

```
public class HourlyEmployee2 extends Employee2 {
    private double hourlyWage;
    private double monthlyHours;

    public HourlyEmployee(String aName, Date aHireDate,
                          double anHourlyWage, double aMonthlyHours) {
        super(aName, aHireDate);
        disallowZeroesAndNegatives(anHourlyWage, aMonthlyHours);
        hourlyWage = anHourlyWage;
        monthlyHours = aMonthlyHours;
    }
    public double getHourlyWage() { return hourlyWage;}
    public double getMonthlyHours() { return monthlyHours;}
    public double getMonthlyPay() { return hourlyWage * monthlyHours; }
    // ...
}
```

Food for thought: what is the monthly pay rule for `HourlyEmployee`s? What if an employee works more than 40 hours per week?

[4] Employee2 is the same as Employee1, but we'll keep the numbers consistent to

Access Restrictions Extend to Subclasses

`private` members of superclasses are present in subclasses, but can't be directly accessed. So this won't compile:

```
public class HourlyEmployee2 extends Employee2 {
  // ...
  public String toString() {
    return name + "; Hire Date: " + hireDate + "; Hourly Wage: "
    + hourlyWage + "; Monthly Hours: " + monthlyHours;
  }
}
```

because `name` and `hireDate` are private in `Employee`. But their getter methods are public:

```
public class HourlyEmployee2 extends Employee2 {
  // ...
  public String toString() {
    return getName()+"; Hire Date: "+getHireDate() +"; Hourly Wage: "
    + hourlyWage + "; Monthly Hours: " + monthlyHours;
  }
}
```

Overriding Methods

Overriding a method means providing a new definition of a superclass method in a subclass. We've been doing this all along with `toString` and `equals`, which are defined in `java.lang.Object`, the highest superclass of all Java classes.

```
public class Object {
    public String toString() {
        return getClass().getName() + "@"
            + Integer.toHexString(hashCode());
    }
    public boolean equals(Object obj) {
        return (this == obj);
    }
}
```

We redefine these on our classes because

- the default implementation of `toString` just prints the class name and hash code (which is the memory address by default).
- the default implementation of `equals` just compares object references, i.e., identity equality, when what we want from `equals` is value equality

`@Override` Annotation

The optional `@Override` annotation informs the compiler that the element is meant to override an element declared in a superclass.

```
public class Employee2 {
  // ...
  @Override
  public String toString() {
    return name + "; Hire Date: " + hireDate;
  }
}
```

Now if our subclass's `toString()` method doesn't actually override `Java.lang.Object`'s (or some other class's) `toString()`, the compiler will tell us.

Programming Exercise

To get some practice writing classes that use inheritance, write:

- A class named `Animal` with:
  - A private instance variable `name`, with a public getter and setter. (Note: `name` is a name of an animal, not the animal's species.)
  - A single constructor that takes the name of the `Animal`
  - A public instance method `speak` that returns a `String` representation of the sound it makes.
- A class named `Dog` that extends `Animal` and specializes the `speak` method appropriately.
- A `Kennel` class with
  - a private instance variable `dogs` that is an array of `Dog`
  - a single constructor that takes a variable number of single `Dog` parameters and initializes the `dogs` instance variable with the constructor's actual parameters.
  - a method `soundOff()` that prints to `STDOUT` (`System.out`) one line for each `Dog` in `dogs` that reads "[dog name] says [output of `speak` method]!", e.g. "Chloe says woof, woof!"

We'll review this at the start of the next lecture