Inheritance

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?

What is inheritance?

More like genetics . . .

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:

```
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.

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) {
        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:[1]

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

    public HourlyEmployee(String name, Date hireDate,
                          double wage, double hours) {
        super(name, hireDate);
        hourlyWage = wage;
        monthlyHours = hours;
    }
    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?

Access Modifiers

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- Every class has an access level (for now all of our classes are `public`).
- Every member has an access level.
- The defulat access level, no mofifier, is also called "package private."

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 `Employee2`. But their getter methods are public:

```
public class HourlyEmployee3 extends Employee3 {
  public String toString() {
    return getName()+", Hire Date: "+getHireDate()
          + ", Wage: "+ hourlyWage
          + ", 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. 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 intermediate superclass's) `toString()`, the compiler will tell us.

Explicit Constructor Invocation with this

What if we wanted to have default values for hourly wages and monthly hours? We can provide an alternate constructor that delegates to our main constructor with this HourlyEmployee3.java:

```
public final class HourlyEmployee3 extends Employee3 {
    /**
     * Constructs an HourlyEmployee with hourly wage of 20
         and
     * monthly hours of 160.
     */
    public HourlyEmployee3(String aName, Date aHireDate) {
        this(aName, aHireDate, 20.00, 160.0);
    }
    public HourlyEmployee3(String aName, Date aHireDate,
                           double anHourlyWage, double
                               aMonthlyHours) {
        super(aName, aHireDate);
        disallowZeroesAndNegatives(anHourlyWage,
            aMonthlyHours);
        hourlyWage = anHourlyWage;
        monthlyHours = aMonthlyHours;
```

this and super

- If present, an explicit constructor call must be the first statement in the constructor.
- Can't have both a super and this call in a constructor.
- A constructor with a this call must call, either directly or indirectly, a constructor with a super call (implicit or explicit).

```
public final class HourlyEmployee3 extends Employee3 {
    public HourlyEmployee3(String aName, Date aHireDate) {
        this(aName, aHireDate, 20.00);
    }
    public HourlyEmployee3(String aName, Date aHireDate,
        double anHourlyWage) {
        this(aName, aHireDate, anHourlyWage, 160.0);
    }
    public HourlyEmployee3(String aName, Date aHireDate,
                           double anHourlyWage, double
                               aMonthlyHours) {
        super(aName, aHireDate);
        disallowZeroesAndNegatives(anHourlyWage,
            aMonthlyHours);
        hourlyWage = anHourlyWage;
```

The Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their supertypes.

Consider the method:

```
public static Date vestDate(Employee employee) {
    Date hireDate = employee.getHireDate();
    int vestYear = hireDate.getYear() + 2;
    return new Date(vestYear,
                    hireDate.getMonth(),
                    hireDate.getDay());
}
```

We can pass any subtype of `Employee` to this method:

```
DateFormat df = DateFormat.getDateInstance();
HourlyEmployee eva = new HourlyEmployee("Eva L. Uator",
                    df.parse("February 13, 2013"), 20.00,
                        200);
Date evaVestDate = vestDate(eva);
```

We must ensure that subtypes are indeed substitutable for supertypes.

LSP Counterexample

A suprising counter-example:

```
public class Rectangle {
  public void setWidth(double w) { ... }
  public void setHeight(double h) { ... }
}
public class Square extends Rectangle {
  public void setWidth(double w) {
    super.setWidth(w);
    super.setHeight(w);
  }
  public void setHeight(double h) {
    super.setWidth(h);
    super.setHeight(h);
  }
}
```

- We know from math class that a square "is a" rectangle.
- The overridden `setWidth` and `setHeight` methods in `Square` enforce the class invariant of `Square`, namely, that `width == height`.

LSP Violation

Consider this client of `Rectangle`:

```
public void g(Rectangle r) {
  r.setWidth(5);
  r.setHeight(4);
  assert r.area() == 20;
}
```

- Client (author of `g`) assumes width and height are independent in `r` becuase `r` is a `Rectangle`.
- If the `r` passed to `g` is actually an instance of `Square`, what will be the value of `r.area()`?

The Object-oriented `is-a` relationship is about behavior. `Square`'s `setWidth` and `setHeight` methods don't behave the way a `Rectangle`'s `setWidth` and `setHeight` methods are expected to behave, so a `Square` doesn't fit the object-oriented is-a `Rectangle` definition. Let's make this more formal . . .

Conforming to LSP: Design by Contract

Require no more, promise no less.

Author of a class specifies the behavior of each method in terms of preconditions and postconditions. Subclasses must follow two rules:

- Preconditions of overriden methods must be equal to or weaker than those of the superclass (enforces or assumes no more than the constraints of the superclass method).
- Postconditions of overriden methods must be equal to or greater than those of the superclass (enforces all of the constraints of the superclass method and possibly more).

In the Rectangle-Square case the postcondition of `Rectangle`'s `setWidth` method:

```
assert((rectangle.w == w) && (rectangle.height ==
    old.height))
```

cannot be satisfied by `Square`, which tells us that a `Square` doesn't satisfy the object-oriented is-a relationship to `Rectangle`.

LSP Conforming 2D Shapes

```
public interface 2dShape {
    double area();
}
public class Rectangle implements 2dShape {
    public void setWidth(double w) { ... }
    public void setHeight(double h) { ... }
    public double area() {
        return width * height;
    }
}
public class Square implements 2dShape {
    public void setSide(double w) { ... }
    public double area() {
        return side * side;
    }
}
```

Notice the use of an interface to define a type.

Interfaces

An interface represents an object-oriented type: a set of public methods (declarations, not definitions) that any object of the type supports. Recall the 2dShape interface:

```
public interface 2dShape {
    double area();
}
```

You can't instantiate interfaces. So you must define a class that implements the interface in order to use it. Implementing an interface is similar to extending a class, but uses the `implements` keyword:

```
public class Square implements 2dShape {
    public void setSide(double w) { ... }
    public double area() {
        return side * side;
    }
}
```

Now a Square is-a 2dShape.

Interfaces Define a Type

```
public interface 2dShape {
    double area();
}
```

This means that any object of type 2dShape supports the area method, so we can write code like this:

```
public double calcTotalArea(2dShape ... shapes) {
    double area = 0.0;
    for (2dShape shape: shapes) {
        area += shape.area();
    }
    return area;
}
```

Two kinds of inheritance: implementation and interface inheritance.

- extending a class means inheriting both the interface and the implementation of the superclass
- implementing an interface means inheriting only the interface, that is, the public methods

Default Methods in Interfaces

Beginning with Java 8 (jdk1.8), interface methods can have default method implementations. These default methods can only reference local variables and static variables defined in the interface or any superinterfaces.

Conflict Resolution for Default Methods

- Superclasses win.
- Interfaces clash.

Static Methods in Interfaces

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.