CS 1331 Introduction to Object Oriented Programming

Inheritance, Part 2 of 2

Christopher Simpkins

chris.simpkins@gatech.edu

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.”

Explicit Constructor Invocation with `this`

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

```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).

```java
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;
        monthlyHours = aMonthlyHours;
    }
```

The Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their supertypes.

Consider the method:

```java
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:

```java
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`:

```java
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

```java
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:

```java
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:

```java
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