




OBJECT-ORIENTED PROGRAMMING

Using Polymorphism Well

Lecture #5

TECHNICAL UNIVERSITY
OF KOSICE



Polymorphism

- Polymorphism means **having many forms**
- How is this related to object-oriented programming?
 - Polymorphic reference
- Since a polymorphic reference can refer to different types of objects over time the specific method it invokes can also change from one invocation to the next

Forms of polymorphism

- **Ad-hoc polymorphism**
 - Polymorphic functions which can be applied to arguments of different types, but which behave differently depending on the type of the argument to which they are applied
 - Function **overloading** (e.g. overloading of constructors and methods)
- **Parametric polymorphism**
 - Allows a function or a data type to be written generically, so that it can handle values identically without depending on their type
 - **Generic programming**
- **Subtype polymorphism** (inclusion polymorphism)
 - Allows a function to be written to take an object of a certain type T , but also work correctly if passed an object that belongs to a type S that is a subtype of T
 - **Subclassing, inheritance**

Overloading

- Providing more methods with the same name but different parameter types
 - Statically select most specific matching method of a type

```
public class Overloaded {
    public String tryMe(Object o) { return "is object"; }
    public String tryMe(String s) { return "is string"; }
    public boolean equals(String s) { return true; }
}
```

Overloads but NOT overrides inherited method **boolean** equals(Object o) from class Object

Overloading

```
public static void main(String[] args) {
    Overloaded over = new Overloaded();

    System.out.println(over.tryMe(over));
    System.out.println(over.tryMe(new String("test")));

    Object obj = new String("test");
    System.out.println(over.tryMe(obj));
    System.out.println(over.equals(new String("test")));
    System.out.println(over.equals(obj));

    Object obj2 = over;

    System.out.println(obj2.equals(new String("test")));
}
```

is object

is string

is object

true

false

false

Overkill

- Overloading (and overriding together) can be overwhelming
- Avoid overloading whenever possible
 - Names are cheap and plentiful
- One place you cannot easily avoid it – constructors
 - They all have to have the same name

Binding

- At a certain point, the method invocation is **bound** to the method definition, that is, a commitment is made to execute certain code
- Though binding often occurs at compile-time, the binding of a method invocation to its definition cannot be made till run-time for a polymorphic reference
- Consider

```
obj.doSomething();
```
- If `obj` is polymorphic, it can refer to different types of objects at different times, thus calling a different definition of `doSomething()` each time it is invoked

Late or dynamic binding

- For polymorphic references, the method definition that is used depends on the object that is being referred by the reference variable at that moment
- This binding decision cannot be made until run-time and is thus called **late** or **dynamic binding**
- Though it is less efficient for bindings to occur at run-time rather than compile-time, it is considered to be an acceptable overhead given the flexibility of a polymorphic reference
- There are two ways to create a polymorphic reference
 - using inheritance
 - using an interface

Overridden method dispatch

- `DiscountArticle` is a subtype of `Article`
- If `DiscountArticle` **overrides** method `getPrice()` of `Article` which method should be called?

```
Article a = new Article();
```

```
DiscountArticle b = new DiscountArticle();
```

```
double p;
```

```
p = a.getPrice();
```

Calls class `Article`'s
`getPrice()` method

```
p = b.getPrice();
```

Calls class `DiscountArticle`'s
`getPrice()` method

```
a = b;
```

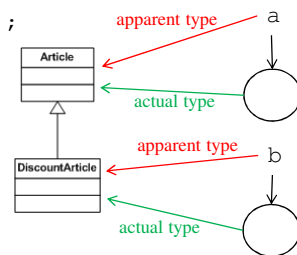
```
p = a.getPrice();
```

Calls class `DiscountArticle`'s
`getPrice()` method

Dynamic dispatch

- Search for the method up the type hierarchy, starting from the actual (dynamic) type of the object

```
Article a = new Article();
DiscountArticle b = new DiscountArticle();
double p;
p = a.getPrice();
p = b.getPrice();
```



Lecture #5: Using Polymorphism Well

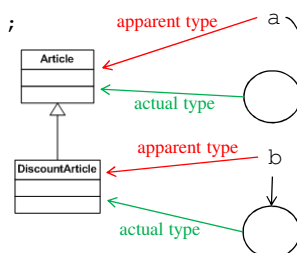
11

Dynamic dispatch

- Search for the method up the type hierarchy, starting from the actual (dynamic) type of the object

```
Article a = new Article();
DiscountArticle b = new DiscountArticle();
double p;
p = a.getPrice();
p = b.getPrice();
a = b;
p = a.getPrice();
```

Now apparent type of a is Article and actual type of a is DiscountArticle



Lecture #5: Using Polymorphism Well

12

Apparent and actual types

- **Apparent types** are associated with declarations
 - They never change
- **Actual types** are associated with object
 - They are always a subtype of the apparent type
- Compiler does type checking using **apparent type**
- During run-time the method dispatch is using **actual type**

Lecture #5: Using Polymorphism Well

Downcasting

- Downcasting (type refinement) is the act of casting a reference of a base class to one of its derived class
 - When a variable of the base class has a value of the derived class, downcasting is possible

```
Article a = new DiscountArticle();
DiscountArticle b = (DiscountArticle) a;
```

This is possible since object referred by a is currently holding value of DiscountArticle class

- The danger of downcasting is that **it is not compile-time check**, but rather **it is run-time check**
 - Downcasting changes apparent type, so **during run-time it must be checked that actual type is a subtype of apparent type**

```
public String objectToString(Object o) {
    return (String) o;
}
```

This will only work when the o currently holding value is String. In compile-time it is OK, because String is subtype of Object

```
String s = objectToString("a test string");
Object wrong = new Object();
s = objectToString(wrong);
```

This will work since we passed in String, so o has value of String

This will fail since we passed in Object which does not have value of String

Lecture #5: Using Polymorphism Well

Downcasting example

- Class `java.util.Vector` has reusable methods

```
public void addElement(Object obj)
public Object elementAt(int i)
```

```
public class StringSet {
    private Vector elements;
    public void insert(String s) {
        elements.addElement(s);
    }
    public String choose() {
        return elements.elementAt(0);
        return (String) elements.elementAt(0);
    }
}
```

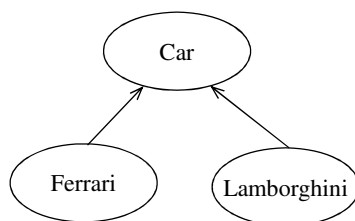
Downcasting is required here

A popular example of a badly considered OO design is containers of universal supertypes, like the Java containers before Java generics were introduced, which requires downcasting of the contained objects so that they can be used again.

Avoid downcasting, since according to the LSP, an OO design that requires it is flawed.

Some languages, such as OCaml, disallow downcasting altogether. In many cases downcasting can be replaced by using **parametric polymorphism** (e.g. generics in C++, C#, Java)

Factoring out variations to the subclasses



```
Car c;
... // c = Ferrari or Lamborghini
if (c instanceof Ferrari)
    ... // draw in red
else
    ... // draw in yellow
... // more car code
```



```
Car c;
... // c = Ferrari or Lamborghini
Color f = c.getColor();
... // more car code
```

Overriding method getColor()

```
public class Car {
    public Color getColor() {
        return Color.BLACK;
    }
}

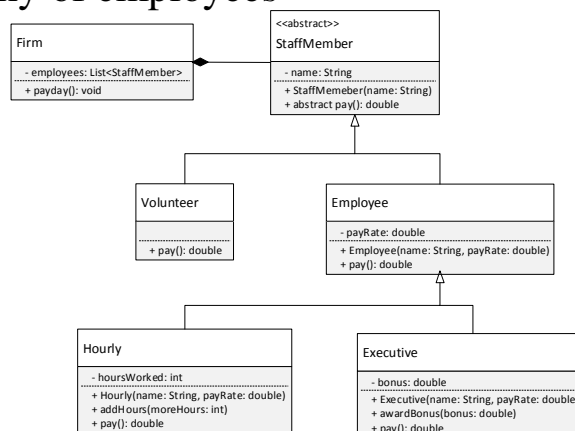
public class Ferrari extends Car {
    @Override
    public Color getColor() {
        return Color.RED;
    }
}

public class Lamborghini extends Car {
    @Override
    public Color getColor() {
        return Color.YELLOW;
    }
}
```

Also abstract class can be used here including abstract method getColor()

Polymorphism via inheritance: An example

■ Hierarchy of employees



TestFirm class definition

- The TestFirm class contains a main driver that creates a Staff of employees and invokes the payday() method

```
public class TestFirm {
    public static void main(String[] args) {
        Firm firm = new Staff();
        firm.payday();
    }
}
```

Firm class definition

- The Firm class maintains an array of StaffMember objects

```
public class Firm {
    private List<StaffMembers> employees;
    public Firm() {
        StaffMember employee1 = new Volunteer("Petra");
        StaffMember employee2 = new Hourly("Jano", 10.55);
        ((Hourly) employee2).addHours(40);
        StaffMember employee3 = new Executive("Robert", 2423.07);
        ((Executive) employee3).awardBonus(500.00);
        this.employees = new Vector<StaffMemeber>();
        this.employees.add(employee1);
        this.employees.add(employee2);
        this.employees.add(employee3);
    }
    public void payday() {
        for (StaffMemeber employee : this.employees) {
            System.out.println(employee);
            double amount = employee.pay();
            if (amount == 0.0)
                System.out.println("Thanks!");
            else
                System.out.println("Paid: " + amount);
            System.out.println("-----");
        }
    }
}
```

The abstract StaffMember class

- The StaffMember class is made abstract because it is generic and likely will not need to be instantiated

```
public abstract class StaffMember {
    private String name;
    public StaffMember(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Name: " + this.name;
    }
    public abstract double pay();
}
```

- The pay() method will differ for each subclass of StaffMember and is therefore made abstract, forcing it to be defined by the subclasses (unless they themselves are abstract)

The Volunteer class

- The Volunteer class inherits from the StaffMember class
- Because it is not an abstract class, it must override the abstract method pay() in StaffMember and provide a definition

```
public class Volunteer extends StaffMember {
    public Volunteer(String name) {
        super(name);
    }
    @Override
    public double pay() {
        return 0.0;
    }
}
```

- Since volunteers are not paid, the return amount of the pay() method is zero

The Employee class

- The `Employee` class also inherits from `StaffMember` and overrides the `pay()` method

```
public class Employee extends StaffMember {
    private double payRate;
    public Employee(String name, double payRate) {
        super(name);
        this.payRate = payRate;
    }
    @Override
    public double pay() {
        return this.payRate;
    }
}
```

- Unlike `StaffMember`, the `Employee` class can be instantiated (even though it too is somewhat generic)

The Hourly class

- The `Hourly` class extends `Employee`, overriding `pay()` method.
- The method `addHours()` is added but cannot be accessed polymorphically (an explicit cast is required) because it is not in other classes of the hierarchy

```
public class Hourly extends Employee {
    private int hoursWorked;
    public Hourly(String name, double payRate) {
        super(name, payRate);
        this.hoursWorked = 0;
    }
    public void addHours(int moreHours) {
        this.hoursWorked += moreHours;
    }
    @Override
    public double pay() {
        double payment = super.pay() * this.hoursWorked;
        this.hoursWorked = 0;
        return payment;
    }
}
```

The Executive class

- The `Executive` class also extend `Employee`
- It provides the additional method `awardBonus()` (which cannot be accessed polymorphically)

```
public class Executive extends Employee {
    private double bonus;
    public Executive(String name, double payRate) {
        super(name, payRate);
        this.bonus = 0;
    }
    public void awardBonus(double bonus) {
        this.bonus = bonus;
    }
    @Override
    public double pay() {
        double payment = super.pay() + this.bonus;
        this.bonus = 0;
        return payment;
    }
}
```

Summary of Firm program

- The `TestFirm` class contains a main driver that creates a `Firm` of employees and invokes the `payday()` method
- The `Firm` class maintains an array of objects that represent individual staff members
- The `employees` list is declared to hold `StaffMember` references, but is actually **filled with objects created from the subclasses**. It is, therefore, filled with **polymorphic references**
- The `payday()` method of the `Firm` class scans through the list of employees, printing their information and invoking their `pay()` methods
- The `pay()` method is **polymorphic** since each class has **its own version**
- The `StaffMember` class is abstract and serves as the ancestor of all the employee classes
- The method `pay()` is abstract and requires a definition by the descendant classes
- The essence of polymorphism: **each class knows best how it should handle a specific behavior**

Polymorphism via interface: An example

- Let's have an interface representing behavior of a fighter
 - Feature hit
- Let's have the classes of fighters
 - Kung-fu fighter class
 - Boxer class



Definition of fighters

```
public interface Fighter {
    public void hit();
}

public class KungFuFighter implements Fighter {
    @Override
    public void hit() {
        System.out.print("trach! ");
    }
}

public class Boxer implements Fighter {
    private boolean nextPunchLeft;
    public Boxer(boolean leftHanded) {
        this.nextPunchLeft = leftHanded;
    }
    @Override
    public hit() {
        System.out.print(this.nextPunchLeft ? "left pow! " : "right pow! ");
        this.nextPunchLeft = !this.nextPunchLeft;
    }
}
```



The FightingArmy class

- Create army of ten various fighters and combat

```
public class Combat {
    public static void main(String[] args) {
        Fighter[] fighters = new Fighter[10];
        for (int i = 0; i < 10; i++)
            if (Math.random() > 0.5)
                fighters[i] = new Boxer(true);
            else
                fighters[i] = new KungFuFighter();
        for (int i = 0; i < 10; i++) {
            System.out.print("Fighter " + (i + 1) + ": ");
            for (int j = 0; j < 5; j++)
                fighters[i].hit();
            System.out.println();
        }
    }
}
```

The Liskov Substitution Principle

- **Functions that use references to base (super) classes must be able to use objects of derived (sub) classes without knowing it**

The Liskov Substitution Principle

- The **Liskov Substitution Principle** (LSP) seems obvious given all we know about polymorphism
- For example:

```
public void drawShape(Shape s) {  
    // Code here  
}
```

- The `drawShape()` method should work with any subclass of the `Shape` superclass (or, if `Shape` is a Java interface, it should work with any class that implements the `Shape` interface)
- But we must be careful when we implement subclasses to ensure that we do not unintentionally violate the LSP

The Liskov Substitution Principle

- If a function does not satisfy the LSP, then it probably makes explicit reference to some or all of the subclasses of its superclass
- Such a function also violates the **Open-Closed Principle**, since it may have to be modified whenever a new subclass is created

LSP example

- Consider the following Rectangle class

```
public class Rectangle {  
    private double width;  
    private double height;  
    public Rectangle(double w, double h) {  
        this.width = w;  
        this.height = h;  
    }  
    public double getWidth() { return this.width; }  
    public double getHeight() { return this.height; }  
    public void setWidth(double w) { this.width = w; }  
    public void setHeight(double h) { this.height = h; }  
    public double area() { return this.width * this.height; }  
}
```

Lecture #5: Using Polymorphism Well

TECHNICAL UNIVERSITY
OF KOŠICE
33

LSP example

- Now, think about a Square class. Clearly, a square is a rectangle, so the Square class should be derived from the Rectangle class
- Observations
 - A square does not need both a width and a height as attributes, but it will inherit them from Rectangle anyway. So, each Square object wastes a little memory, but this is not a major concern
 - The inherited `setWidth()` and `setHeight()` methods are not really appropriate for a Square, since the width and height of a square are identical. So we'll need to override `setWidth()` and `setHeight()`. (Having to override these very basic methods is a clue that this might not be an appropriate use of inheritance!)

Lecture #5: Using Polymorphism Well

TECHNICAL UNIVERSITY
OF KOŠICE
34

LSP example

- Here's the Square class

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

LSP example

- Everything looks good, but check this out!

```
public class TestRectangle {
    public static void testLSP(Rectangle r) {
        r.setWidth(4.0);
        r.setHeight(5.0);
        System.out.println("Width is 4.0 and Height is 5.0" + ", so Area is " +
            r.area());
        if (r.area() == 20.0)
            System.out.println("Looking good!\n");
        else
            System.out.println("Huh? What kind of rectangle is this?\n");
    }
    public static void main(String[] args) {
        Rectangle r = new Rectangle(1.0, 1.0);
        Square s = new Square(1.0);
        testLSP(r);
        testLSP(s);
    }
}
```

Now call the method `testLSP()`. According to the LSP, it should work for either rectangles or squares. Does it?

Remember: If *B* is a subtype of *A*, everywhere the code expects an *A*, a *B* can be used instead.

LSP example

- Test program output

```
Width is 4.0 and Height is 5.0, so Area is 20.0  
Looking good!
```

```
Width is 4.0 and Height is 5.0, so Area is 25.0  
Huh? What kind of rectangle is this?
```

- Looks like we violated the LSP!

LSP example

- What is the problem here? The programmer of the `testLSP()` method made the reasonable assumption that changing the width of a `Rectangle` leaves its height unchanged
- Passing a `Square` object to such a method results in problems, exposing a violation of the LSP
- The `Square` and `Rectangle` classes look self consistent and valid. Yet a programmer, making reasonable assumptions about the base class, can write a method that causes the design model to break down
- Solutions can not be viewed in isolation, they must also be viewed in terms of reasonable assumptions that might be made by users of the design

LSP example

- A mathematical square might be a rectangle, but a Square object is not a Rectangle object, because the behavior of a Square object is not consistent with the behavior of a Rectangle object!
- **Behaviorally**, a Square is **not** a Rectangle!
A Square object is **not** polymorphic with a Rectangle object

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) makes it clear that the IS-A relationship is all about **behavior**
- In order for the LSP to hold (and with it the Open-Closed Principle) all subclasses must conform to the behavior that clients expect of the base classes they use
- A subtype must have no more constraints than its base type, since the subtype must be usable anywhere the base type is usable
- If the subtype has more constraints than the base type, there would be uses that would be valid for the base type, but that would violate one of the extra constraints of the subtype and thus violate the LSP!
- **The guarantee of the LSP is that a subclass can always be used wherever its base class is used!**

Substitution principle guarantee

- **How do we know if saying B is a subtype of A is safe?**
- If B is a subtype of A , everywhere the code expects A , a B can be used instead
- For function $f(A)$, if f satisfies its specification when passed an object whose actual type is type A , f also satisfies its specification when passed an object whose type is B

Subtype condition 1: Signature rule

```

class A {
  public  $R_A.f(P_A p)$ ;
}
class B extends A {
  public  $R_B.f(P_B p)$ ;
}
  
```

- R_B must be a subtype or R_A ($R_B \leq R_A$)
 - Covariants for results
- P_B must be a supertype of P_A ($P_A \leq P_B$)
 - Contravariants for parameters

Subtype condition 1: Signature rule

- Java compiler (before version 1.5) is stricter than this, it does not allow any variations in types (no variant)
 - Overriding method must have same return and parameter types
 - Overriding method can throw fewer exceptions
- Before versions use overloading in this case, so the methods are not actually overridden
- Later versions use `@Override` clause to specify the variant method overrides the method from the superclass

Subtype condition 2: Methods rule

- Pre condition of the subtype method must be **weaker** than the pre condition of the supertype method ($pre_A \Rightarrow pre_B$)
 - The rectangle must have its width and height set to calculate its area
 - The square must have its width and height set and they must be the same to calculate its area
 - This means square has **stronger pre condition** for calculating area than rectangle. **FAIL**
- Post condition of the subtype method must be **stronger** than the post condition of the supertype method ($post_B \Rightarrow post_A$)
 - The article gets actual price as its normal sales price
 - The discount article gets actual price as its normal sales price updated by the percentage discount
 - This means discount article has **stronger post condition** for getting actual price. **OK**

Subtype condition 3: Properties

- Subtypes must preserve all properties described in the overview specification of the supertype
- Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be provable for objects y of type S where $S \leq T$
- $(pre_{super} \wedge post_{sub}) \Rightarrow post_{super}$
- Example: Let $B \leq A$
 - If A is immutable data type, can B be mutable?
 - If A is mutable data type, can B be immutable?

Eiffel's rule

- Another approach to type substitution
- Bertrand Meyer in OO language Eiffel prefers **covariant** typing
 - The subtype replacement method parameter types must be **subtypes** of the types of the parameters of the supertype method
- The subtype method pre conditions must be stronger than the supertype method pre condition and the subtype post conditions must be stronger than the supertype post conditions
- Note that unlike the corresponding Liskov substitution principle, $(pre_{super} \wedge post_{sub}) \Rightarrow post_{super}$, there is no need for pre_{super} in the covariant rule since $post_{sub} \Rightarrow post_{super}$

Readings

- MEYER, B.: *Object-Oriented Software Construction*. 2nd Edition, Prentice Hall, 1997
 - Chapter 24: Using Inheritance Well