Q: Why do Java developers wear glasses?

A: Because they do not C#!

OBJECT-ORIENTED PROGRAMMING

Inheritance and Polymorphism

Lecture #3

TECHNICAL UNIVERSITY OF KOŠICE

Motivation

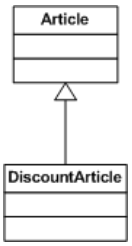
- Interesting systems are seldom born into an empty world
  - Almost always, new software expands on previous developments; the best way to create it is by imitation, refinement and combination
  - Traditional design methods largely ignored this aspect of system development
- In object technology it is an essential concern

Motivation

- Reusability
  - To avoid rewriting the same code over and over again, wasting time, introducing inconsistencies and risking errors, we need techniques to capture the striking commonalities that exist within groups of similar structures – all text editors, all tables, all file handlers – while accounting for the many differences that characterize individual cases
- Extendibility
  - The type system in traditional programming languages has the advantage of guaranteeing type consistency at compile time, but prohibits combination of elements of diverse forms even in legitimate cases

Subtyping

- Remember our Shop Project from the previous lectures
  - Now, let's use new article type with percentage discount and changed or new behavior
- `DiscountArticle` is a **subtype** of `Article` and `Article` is a **supertype** of `DiscountArticle`
- `DiscountArticle` ≤ `Article`



Subtype substitution

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

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

`b` must be a subtype of `Article` (note *A* is subtype of *A*)

Implementation issues

- Subtyping
  - Allow one type to be used where another type is expected
- Inheritance
  - Reuse implementation of the supertype to implement a subtype

Inheritance

- To implement a subtype, it is often useful to use the implementation of its supertype
- This is also called **subclassing**
- In Java:

```
class B extends A
```

Both subtyping and inheritance

`B` is a subtype of `A`

`B` inherits from `A`

```
class C implements D
```

Just subtyping

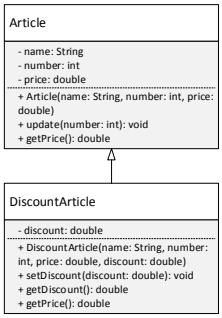
`C` is a subtype of `D`

Inheritance example: Analysis

- Remember our Shop Project from the previous lectures
  - We have `Shop` and `Article` classes defined to implement articles selling process
- Now, let's use new article type with percentage discount and changed or new behavior
  - New discount article will use the same characteristics and behavior as normal articles
  - There are also new characteristics and extended behavior for the discount article
    - Percentage discount value
    - Article price calculation must apply the discount

Inheritance example: Design

- By inheritance we can create new classes based on existing classes
  - New class `DiscountArticle` inherits all characteristics and behaviors of class `Article` and also defines some new



Inheritance example: Implementation

- The code for inherited class `DiscountArticle`

```
public class DiscountArticle extends Article {
  private double discount;

  public DiscountArticle(String name, int number, double price, double discount) {
    super(name, number, price);
    this.discount = discount;
  }

  public double getDiscount() {
    return this.discount;
  }

  public void setDiscount(double discount) {
    this.discount = discount;
  }

  @Override
  public double getPrice() {
    return super.getPrice() * (1.0 - this.discount);
  }
}
```


Inheritance example: Test

- Test code

```
public class ShopApp {
  public static void main(String[] args) {
    List<Article> articleList = new Vector();

    Article article1 = new Article("Gin", 100, 10.25);
    Article article2 = new Article("Whisky", 150, 12.40);
    Article article3 = new DiscountArticle("Fernet", 200, 10.0, 0.2);
    artilceList.add(article1);
    artilceList.add(article2);
    artilceList.add(article3);

    Shop shop = new Shop("Alco", articleList);

    shop.sell(article3, 1);

    System.out.println("Shop turnover is " + shop.getTurnover());
  }

}
```

Reuse of function `sell()`. We do not need to rewrite the functionality of article selling, it remains the same.

- Test output

```
Shop turnover is 8
```

Terminology remarks

- `Article`
  - Main class, parent class, ancestor, superclass
- `DiscountArticle`
  - Inherited class, derived class, child (daughter) class, descendant, subclass
- Inheritance is also called **subclassing**

Using inheritance

- We can define new classes easily, we do not need to rewrite whole behavior
- We can use new object-oriented notion of types – **polymorphism**
  - What type is object `article3` of in our example?
    - The reference to the object `article3` is of type `Article`, but the object itself is from class `DiscountArticle`
    - Object `article3` can be used both as `Article` and `DiscountArticle` types
  - Objects from subclasses can be always used as objects from their superclasses

Using inheritance

- Derived classes can **define new** characteristics and behaviors
  - New characteristics and behaviors can be used only with derived class
  - For example in our example we cannot call `article3.getDiscount()`, because the reference `article3` is of type `Article` and `getDiscount()` is behavior of `DiscountArticle`
  - We must take care what types do we use (superclass or subclass?) with the object
- Derived classes can **rewrite (override) existing** behaviors
  - Overridden behaviors are always used according the creation class of the object
  - For example in our example we can call `article3.getPrice()`, even if the reference to the `article3` object is of type `Article`, the behavior from class `DiscountArticle` is used
  - Note: OO languages (Java, C++, C# etc.) have various possibilities to rewrite functions in subclasses

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

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

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

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

Information hiding in inheritance

- All characteristics and behaviors are inherited to new class
- We can use accessibility modificators to control the **direct accessibility** of inherited properties
  - **`public`** – properties are accessible from outer classes and all derived classes
  - **`private`** – properties are accessible only in the class it is defined
  - **`protected`** – properties are not accessible from outer classes, but still accessible from derived classes

Constructing object of derived class

- At the beginning of each constructor of subclass **the default constructor of superclass is called**
  - In our example when creating object of `DiscountArticle` class the default `Article()` constructor within `DiscountArticle()` constructor is called
  - When we want to call other than default constructor we can use **`super`** keyword to call different overloaded version of constructor, as we can see in our example

```
public DiscountArticle(String name, int number, double price, double discount) {
  super(name, number, price);
  this.discount = discount;
}
```

Using the `super` keyword

- Accessing overridden superclass members
  - If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`

```
public double getPrice() {
  return super.getPrice() * (1.0 - this.discount);
}
```

- Subclass constructor
  - As mentioned before when we want to invoke superclass's constructor

```
public DiscountArticle(String name, int number, double price, double discount) {
  super(name, number, price);
  this.discount = discount;
}
```


Methods of reuse

- Another way of functionality reuse is **objects composition**
- One of the main object-oriented principle is to **favor composition over inheritance**
- Both composition and inheritance support
  - Reusability
  - Extendability

Composition

- Method of reuse in which new functionality is obtained by creating an object **composed of** other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- Sometimes called **aggregation** or **containment**, although some authors give special meanings to these terms
  - **Aggregation** – when one object owns or is responsible for another object and both objects have identical lifetimes or when one object has a collection of objects that can exist on their own
  - **Containment** – a special kind of composition in which the contained object is hidden from other objects and access to the contained object is only via the container object

Implementation of composition

- Composition can be
  - By reference
  - By value
- C++ allows composition by value or by reference
- But in Java or C# all we have are object references

Advantages/disadvantages of composition

- Advantages
  - Contained objects are accessed by the containing class solely through their interfaces
  - "Black-box" reuse, since internal details of contained objects are **not** visible
  - Good encapsulation
  - Fewer implementation dependencies
  - Each class is focused on just one task
  - The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type
- Disadvantages
  - Resulting systems tend to have more objects
  - Interfaces must be carefully defined in order to use many different objects as composition blocks

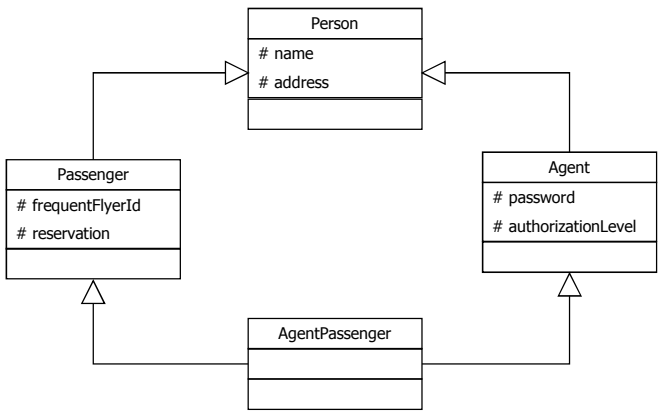
Advantages/disadvantages of inheritance

- Advantages
  - New implementation is easy, since most of it is inherited
  - Easy to modify or extend the implementation being reused
- Disadvantages
  - Breaks encapsulation, since it exposes a subclass to implementation details of its superclass
  - "White-box" reuse, since internal details of superclasses are often visible to subclasses
  - Subclasses may have to be changed if the implementation of the superclass changes
  - Implementations inherited from superclasses can not be changed at runtime

Coad's rules

- Use inheritance only when all of the following criteria are satisfied
  - A subclass expresses **is a special kind of** and not **is a role played by a**
  - An instance of a subclass never needs to become an object of another class
  - A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
  - A subclass does not extend the capabilities of what is merely a utility class
  - For a class in the actual Problem Domain, the subclass specializes a role, transaction or device

Inheritance vs composition: Example 1

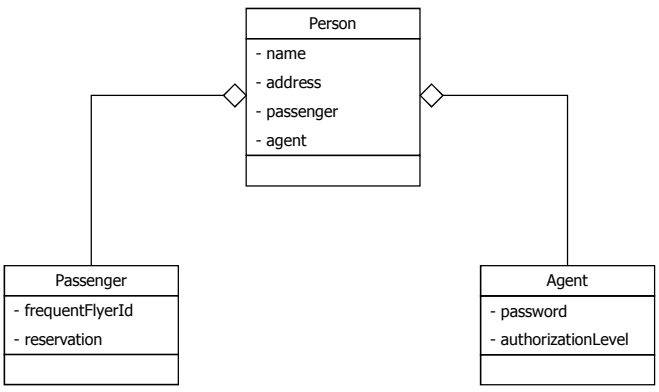


Inheritance vs composition: Example 1

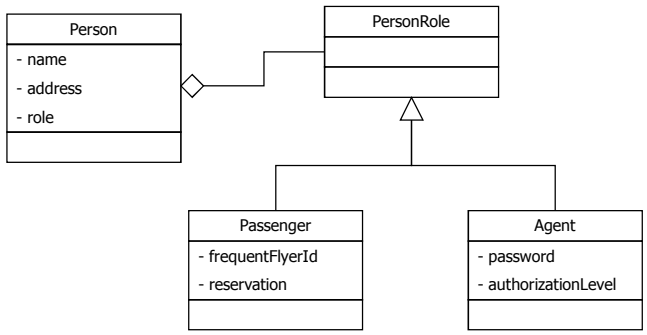
- "Is a special kind of" not "is a role played by a"
  - **Fail**. A passenger is a role a person plays. So is an agent.
- Never needs to transmute
  - **Fail**. A instance of a subclass of Person could change from Passenger to Agent to Agent Passenger over time
- Extends rather than overrides or nullifies
  - **Pass**.
- Does not extend a utility class
  - **Pass**.
- Within the Problem Domain, specializes a role, transaction or device
  - **Fail**. A Person is not a role, transaction or device.

- **Inheritance does not fit here!**

Inheritance vs composition: Example 1



Inheritance vs composition: Example 2



Inheritance vs composition: Example 2

- "Is a special kind of" not "is a role played by a"
  - **Pass**. Passenger and agent are special kinds of person roles.
- Never needs to transmute
  - **Pass**. A Passenger object stays a Passenger object; the same is true for an Agent object.
- Extends rather than overrides or nullifies
  - **Pass**.
- Does not extend a utility class
  - **Pass**.
- Within the Problem Domain, specializes a role, transaction or device
  - **Pass**. A PersonRole is a type of role.

- **Inheritance ok here!**

Inheritance vs composition summary

- Both composition and inheritance are important methods of reuse
- Inheritance was overused in the early days of OO development
- Over time we have learned that designs can be made more reusable and simpler by favoring composition
- Of course, the available set of composable classes can be enlarged using inheritance
- So composition and inheritance work together
- But our fundamental principle is
  - **Favor composition over inheritance**

Readings

- ECKEL, B.: *Thinking in Java*. 4th Edition, Prentice Hall, 2006
  - Reusing Classes (pp. 165 – 192)
  - Polymorphism (pp. 193 – 218)