

Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner Dietl

Software Component Technology

Exercises 4: Components

Wintersemester 05/06

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Core Concepts: Summary

- Core concepts of the OO-paradigm
 - Object model
 - Interfaces and encapsulation
 - Classification and polymorphism
- Core concepts are **abstract concepts** to meet the new requirements
- To apply the core concepts we need ways to **express them in programs**
- **Language concepts** enable and facilitate the application of the core concepts

Subtyping

- **Substitution principle**

Objects of subtypes can be used wherever objects of supertypes are expected

- **Subtype polymorphism**

Program parts working with supertype objects work as well with subtype objects

Inheritance versus Subtyping

- **Subtyping** expresses **classification**
- **Inheritance** is a means of **code reuse**
- Inheritance is **usually coupled** with subtyping
 - Terminology: **Subclassing** = Subtyping + Inheritance
- Issues
 - Subtyping without inheritance
 - Inheritance without subtyping

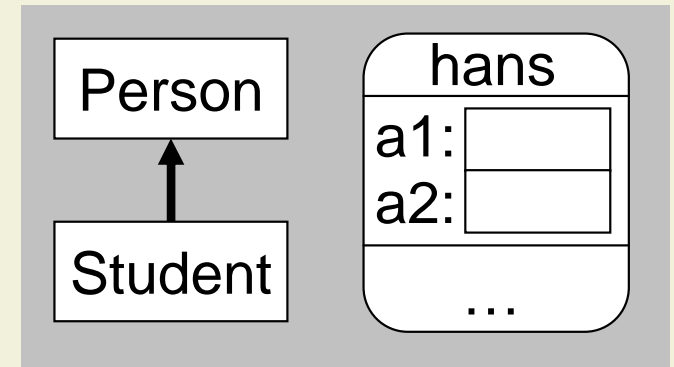
Rules for Subtyping: Summary

- Subtype objects must **fulfill contracts** of supertypes, but
 - Subtypes can have **stronger invariants**
 - Overriding methods of subtypes can have **weaker preconditions**
stronger postconditions than corresponding supertype methods
- Concept is called **Behavioral Subtyping**
- Consequence of substitution principle

Main Forms of Reuse “in the Small”

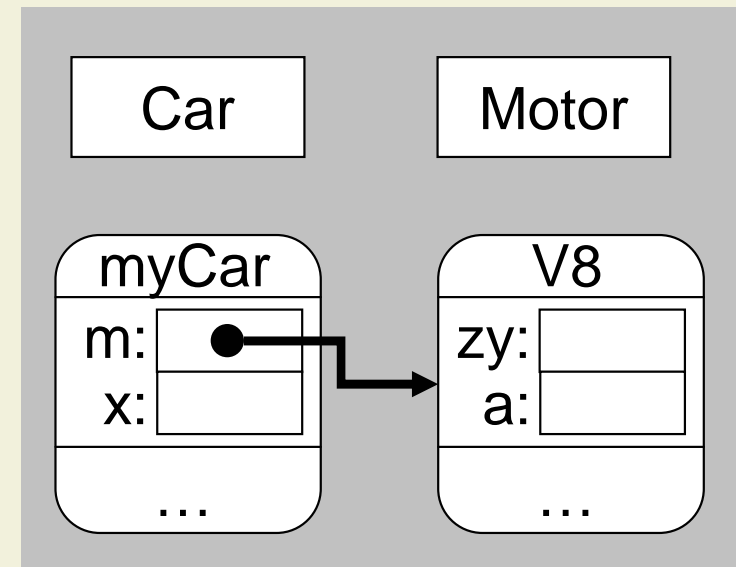
■ Inheritance

- Subclassing establishes **“is-a” relation**
- Enables subtype **polymorphism**
- Only **one object** at runtime



■ Aggregation

- Establishes **“has-a” relation**
- **No subtyping** in general
- **Two objects** at runtime



Homework – Exercise 6 – Set

```
public void add( Object o ) {  
    assert true : "Any state allowed."; // pre  
    ...  
    assert isElem( o ) : "Element not  
        successfully inserted!"; // post  
}
```

```
public void remove( Object o ) {  
    assert isElem( o ) : "Element must be  
        contained in the set!"; // pre  
    ...  
    assert !isElem( o ) : "Element was not  
        removed successfully!"; // post  
}
```

Homework – Exercise 6 – Set

```
public boolean isElem( Object o ) {  
    assert true :  
        "Any state allowed."; // pre  
    ...  
}
```

```
public int size() {  
    assert true : "Any state allowed."; // pre  
    ...  
    assert s == set_impl.size() :  
        "Inconsistent size value!"; // post  
}
```


Homework – Exercise 6 – BoundedSet

```
public void add( Object o ) {  
    assert size() < max : "Size >= max!"; // pre  
    ...  
    assert isElem( o ) : "Element not successfully  
        inserted!"; // post  
    assert size() <= max : "Size > max!"; // invariant  
}  
  
public void remove( Object o ) {  
    assert isElem( o ) : "Element must be contained in  
        the set!"; // pre  
    ...  
    assert !isElem( o ) : "Element was not removed  
        successfully!"; // post  
    assert size() <= max : "Size > max!"; // invariant  
}
```

Homework – Exercise 6 – BoundedSet

```
public boolean isElem( Object o ) {  
    assert true : "Any state allowed."; // pre  
    ...  
    assert size() <= max : "Size > max!"; // invariant  
}  
  
public int size() {  
    assert true : "Any state allowed."; // pre  
    ...  
    assert s == set_impl.size() : "Inconsistent size  
        value!"; // post  
    assert s <= max : "Size > max!"; // invariant  
}
```

Homework – Exercise 7

- ObjectList:

```
// requires type( obj ) <= Object  
void insert( Object obj ) ...
```

- IntegerList:

```
// requires type( obj ) <= Integer  
void insert( Object obj ) ...
```

➔ IntegerList no subtype of ObjectList,
because the precondition gets stronger!

Homework – Exercise 7

- So should `ObjectList` be a subclass of `IntegerList`?
 - Contract of the `insert` method would get weaker!
 - Integerlist:

```
// ensures type( res ) <= Integer  
Object get( int idx ) ...
```
 - ObjectList:

```
// ensures type( res ) <= Object  
Object get( int idx ) ...
```
- ➔ Postcondition gets weaker, which is not allowed!

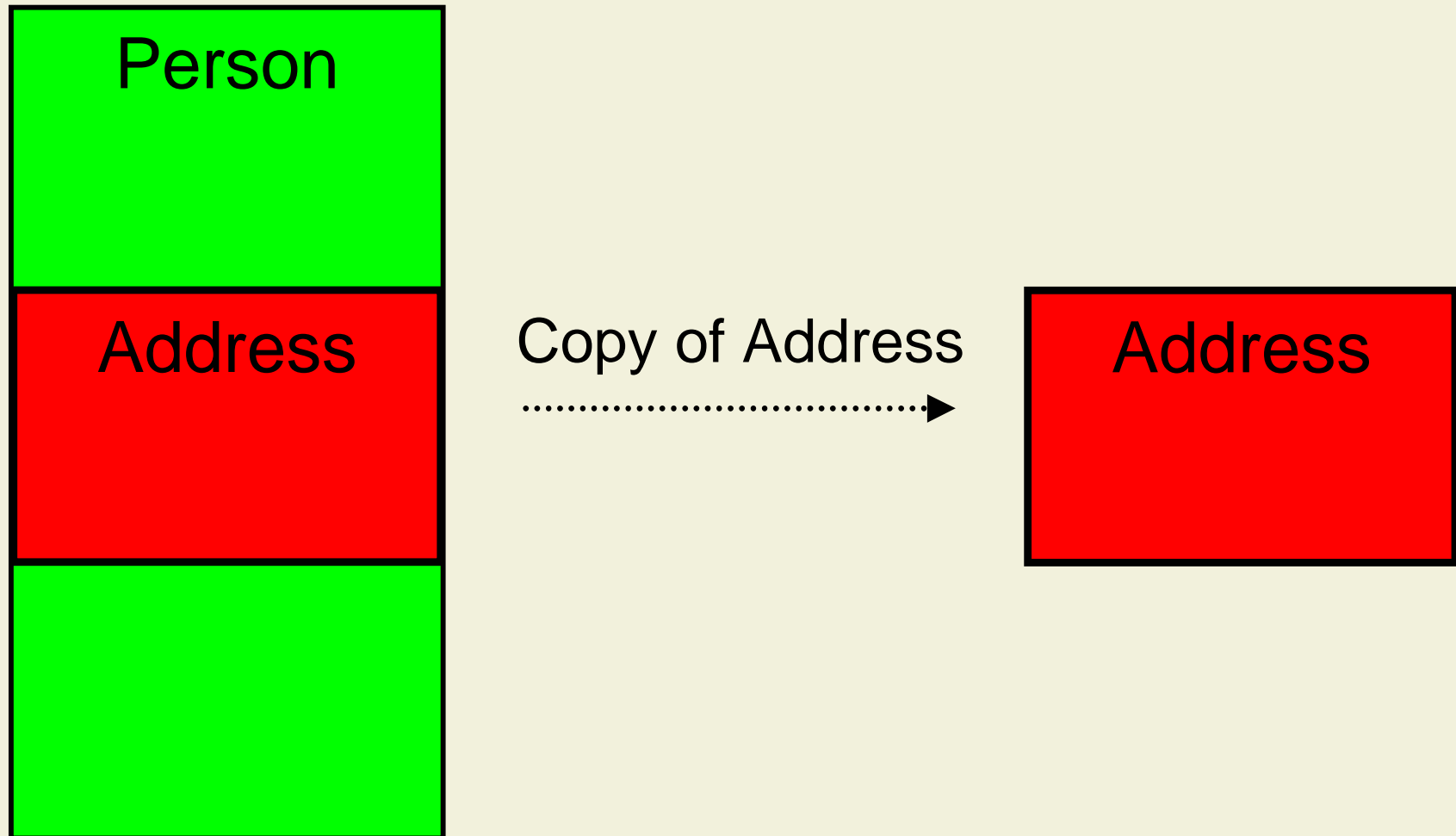
Homework – Exercise 8

- `List<Object>` substituted method signatures:
`void insert(Object obj)`
`Object get(int idx)`
- `List<Integer>` substituted method signatures:
`void insert(Integer obj)`
`Integer get(int idx)`
- As before, the two possible subtype relationships would violate behavioral subtyping
- Java forbids subtype
- Eiffel does not check and might create a CAT call

Aggregation

- Java always stores references to other objects
- In e.g. C++ the developer has the choice to either save a reference to the aggregate object or include it in its own body
- “has-a” and “part-of” aggregation
- Let’s look at the pros and cons a bit
- `Address` and `FullAddress` store information
- `Address > FullAddress`
- A `Person` uses an `Address` object to store its information

Part-of Aggregation



Part-of Aggregation in C++

```
class Person {  
    Person(...) { ... }  

```

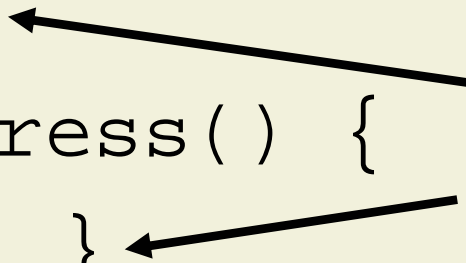
```
    void setAddress( const Address &a ) {
```

```
        addr = a; }
```

```
    Address getAddress() {
```

```
        return addr; }
```

Copy-Constructor of
Address is called!



```
    Address addr;
```

```
} ;
```


Problem with Part-of Aggregation

```
Person p( ... );
```

```
FullAddress a( ... );
```

```
cout << "Person after construction: " << endl;
```

```
cout << p.getAddress().getID() << endl;
```

```
p.setAddress( a );
```

Only the Address information is copied!
System not extendible by subtyping!

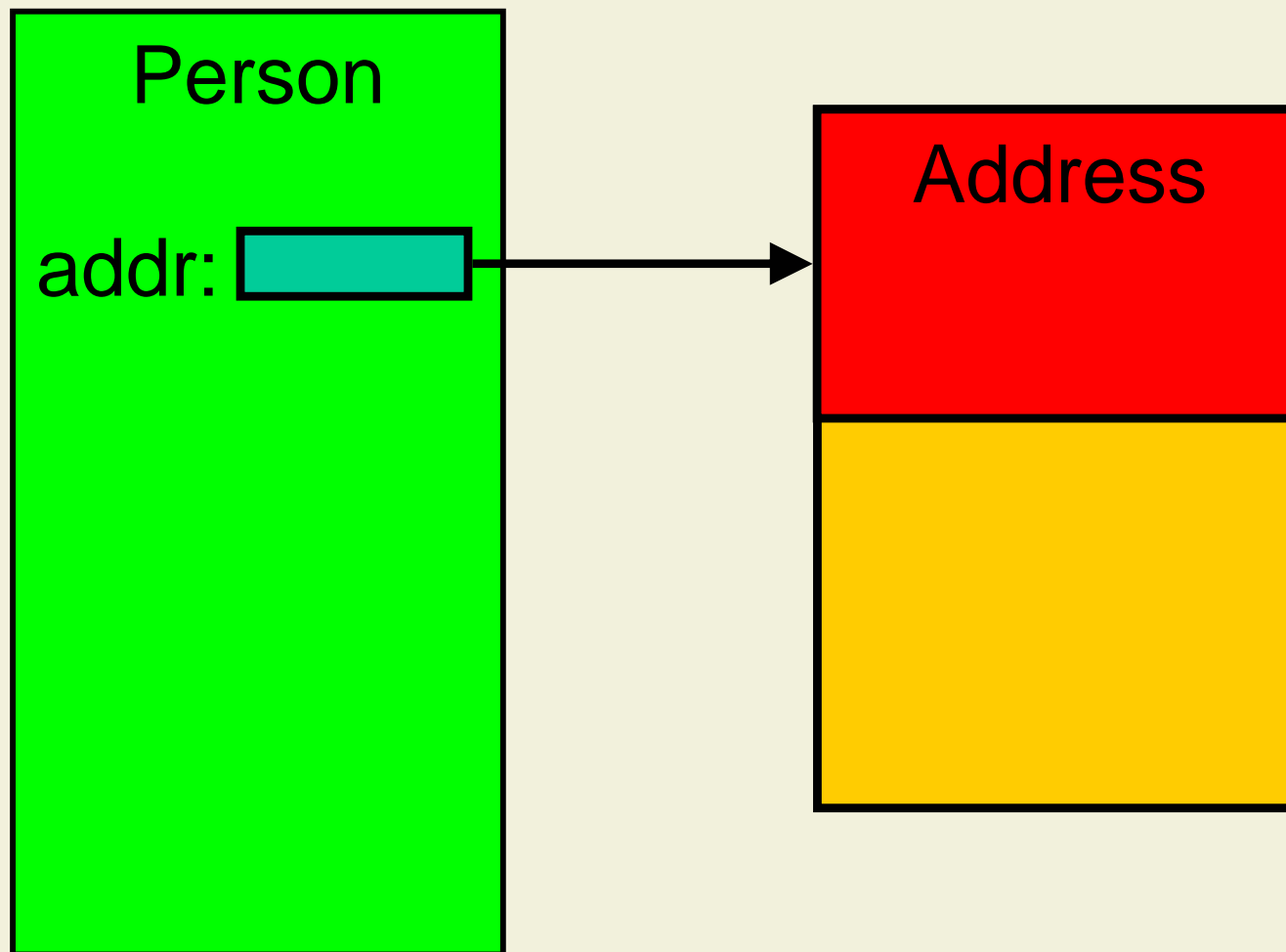
```
cout << "Person after asssigning a  
FullAddress: " << endl;
```

```
cout << p.getAddress().getID() << endl;
```

Advantages of Part-of Aggregation

- It is a bit harder for the programmer to create unwanted aliases to the aggregate
- It might be more efficient

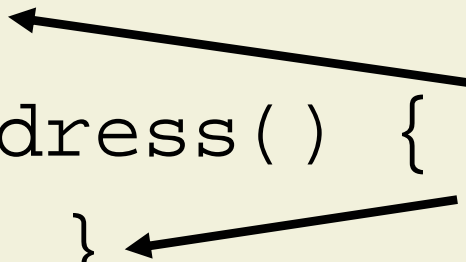
Has-a Aggregation



Has-a Aggregation in C++ (Always in Java)

```
class Person {  
    Person(...) { addr = new Address(...); }  
  
    void setAddress( Address *a ) {  
        addr = a; }  
    Address *getAddress( ) {  
        return addr; }  
};
```

Reference to Address
is stored/returned!



Pros & Cons of Has-a Aggregation

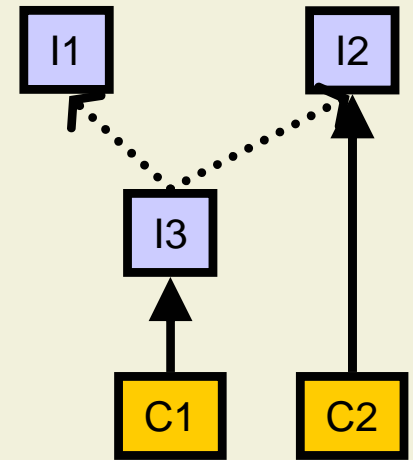
- The aggregate can be exchanged at runtime with a different object, even of a different subtype
- Dynamic change of behavior possible
- Recursive data structures like linked-lists only possible with has-a aggregation
- Problems with multiple references to the same object difficult to handle

Another Casting Example

```
interface I1 {}           interface I2 {}  
interface I3 extends I1, I2 {}
```

```
class C1 implements I3 {}  
class C2 implements I2 {}
```

```
class Conv {  
    public static void main( String argv[] ) {  
1      C1 c1 = new C1();  
2      I2 i2 = (I2) c1;  
3      String s = (String) i2;  
4      C2 c2 = (C2) i2;  
5      I1 i1 = (I1) i2;  
6      I3 i3 = new I3();  
7      i1 = (I1) c1;  
    }  
}
```



Compilation Error!
ClassCastException

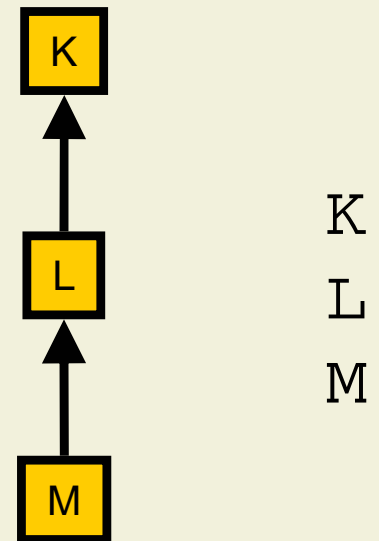
Compilation Error!

What's this programs output?

```
class K {  
    K() { System.out.println("K"); }  
  
    public static void main(String[] args) { new M(3); }  
}
```

```
class L extends K {  
    L() { System.out.println("L"); }  
}
```

```
class M extends L {  
    M(int p) { System.out.println("M"); }  
}
```



What's this programs output?

```
class K {  
    K(int p) { System.out.println("K"); }  
  
    public static void main(String[] args) { new M(3); }  
}  
  
class L extends K {  
    L(int p) { System.out.println("L"); }  
}  
  
class M extends L {  
    M(int p) { System.out.println("M"); }  
}
```

Compilation Error!

Dynamic Type Information

```
class A {  
    void m( ) {  
        if( this instanceof A )  
            System.out.println( "I'm an A!" );  
        if( this instanceof B )  
            System.out.println( "I'm a B!" );  
    }  
}
```

```
class B extends A {  
    public static void main(String[] args) {  
        A a = new B( );  
        a.m( );  
    }  
}
```

I'm an A!
I'm a B!

Object Comparison

```
class D {  
    private int x;  
    public D( int p ) { x = p; }  
  
    public boolean equals( Object o ) {  
        if( o instanceof D ) {  
            return ((D)o).x == this.x;  
        } else {  
            return false;  
        }  
    }  
}
```

Questions?