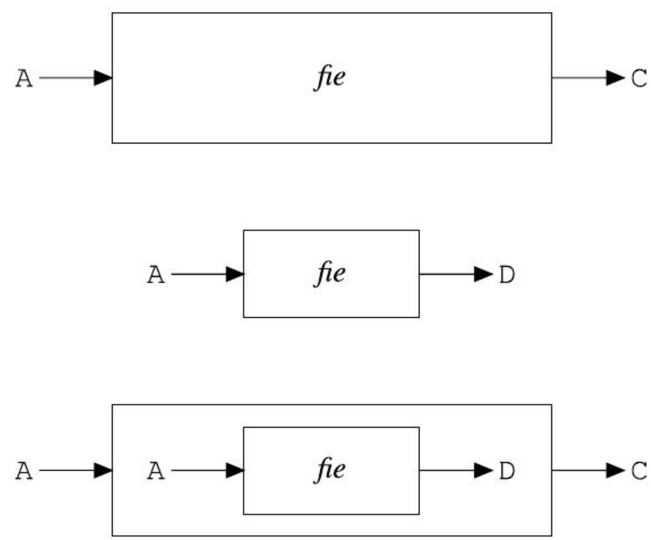
Object Orientation: covariance and implementation

Based on the slides of Maurizio Gabbrielli

Covariance and Contravariance

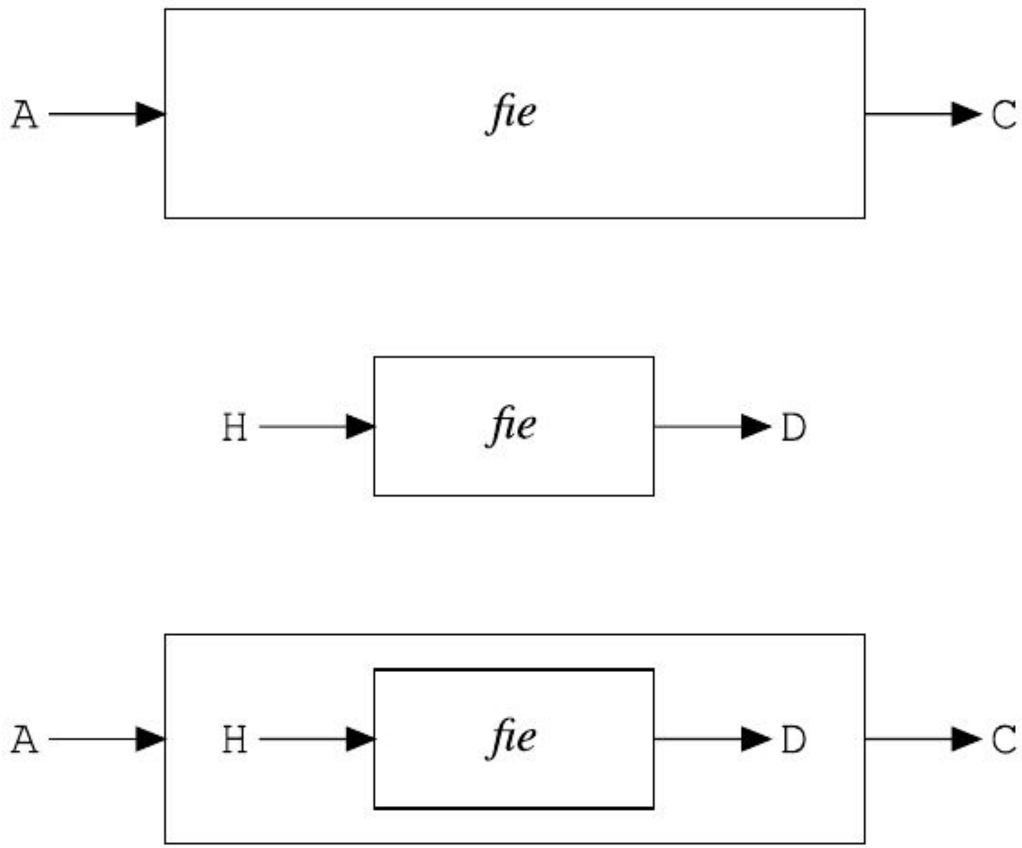
- How subtyping between more complex types relates to subtyping between their components
- Covariance
  - B <: A (B subtype of A) implies that T(B) <: T(A)
- If D <: C (D subtype of C) then overriding is correct
- Overriding correct when covariant on result (E <: F)

```
class F{
    C fie (A p) {...}
}

class E extending F{
    D fie (A p) {...}
}
```



Covariance and Contravariance

- More general if A <: H and D <: C
- Contravariant overriding of methods arguments OK



Contravariance of method arguments

- Input of overridden methods need to be contravariant

```
class Point{
   ...
   boolean eq(Point p){...}
}
```

```
class ColoredPoint extending Point{
   ...
   boolean eq(ColoredPoint p){...}
}
```

- `ColoredPoint` not sublcass of `Point` because argument of eq subtype (not supertype) of `Point`!
- Contravariant overriding is often not used (Java, C++ require type identity for args)
- Some languages used covariant args overriding (semantically wrong!, e.g. Eiffel)

Java not preserving Type Hierarchy Subtyping

- Java does not preserve type hierarchy subtyping
- If A <: B XXX<A> is not related to XXX<B>

```
Stack<Integer> si = new Stack<Integer>();
Stack<Object> so = si;                          // Incorrect in  Java
```

Otherwise it will be possible to do

```
so.push(new String("pluto"));
Integer i = si.pop();                          // danger!
```

Java and covariant arrays

- B <: A implies that B[] <: A[]
- Is that the right choice?

```
class A {int x;}
class B extends A {int y;}

B[] bvect = new B[10];
A[] avect = bvect;      // OK because B[] <: A[]
avect[0] = new A();      // allowed by compiler
                         // but error at run-time
                         // raises ArrayStoreException
```

Note: there is a long discussion on covariant, contravariant, and invariant. Java is invariant on the arguments, covariant on the returned value. Alternatives are possible: Eiffel has unsafe covariant + covariant, Sather has safe contravariant + covariant

Implementation Aspects: Objects

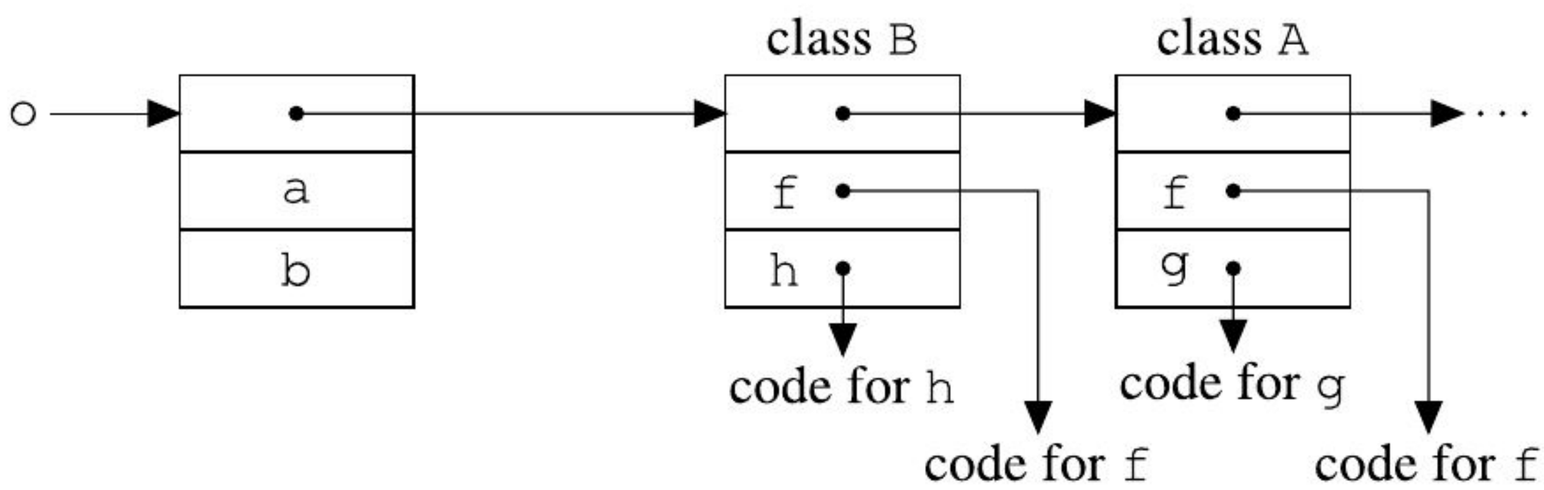
- Each object is represented as a record
  - As many fields as variables of the class + variables of superclasses
  - In case of ' ' shadowing ' ' more fields for the same name
  - Pointer to the class descriptor
- In the case a static type system is used:
  - known at compile time the offset of each field relative to the beginning of the memory record for the object

Implementation Aspects: Classes

- Simplest case: concatenated list
- The invocation of a method on an object accesses the class and then goes up the hierarchy (list)
  - Adopted in Smalltalk: simple but inefficient solution

Example

```
class A{
    int a;
    void f(){...}
    void g(){...}
}
class B extending A{
    int b;
    void f(){...} // redefined
    void h(){...}
}
B o = new B();
```

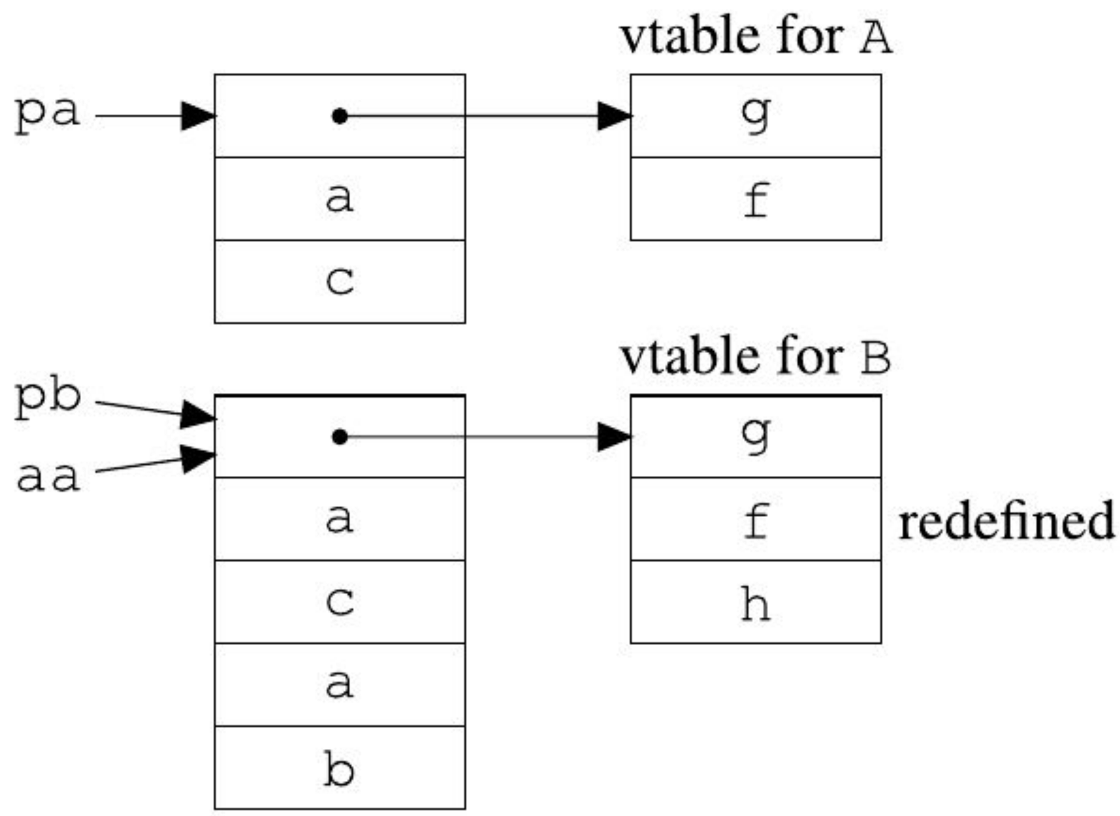


A better solution for inheritance

- Prerequisite: static type system → because in this way set of methods that an object can invoke is know as compile time
- The list of all methods is retained in the class descriptor in a Virtual function table (V-table)
  - Each class has its own V-table
- If A subclass of B, the V-table of A contains in the initial part the one of B
  - Invocation of a method requires two indirect accesses, without scanning

Implementing Inheritance with V-table

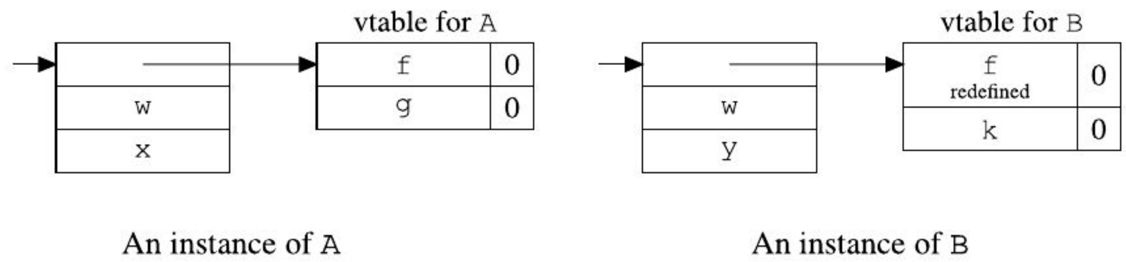
```
class A{
    int a;
    char c;
    void g(){...}
    void f(){...}
}
class B extending A{
    int a;
    int b;
    void h(){...}
    void f(){...}
        // redefined

}
B pb = new B();
A pa = new A();
A aa = pb;
aa.f();
```

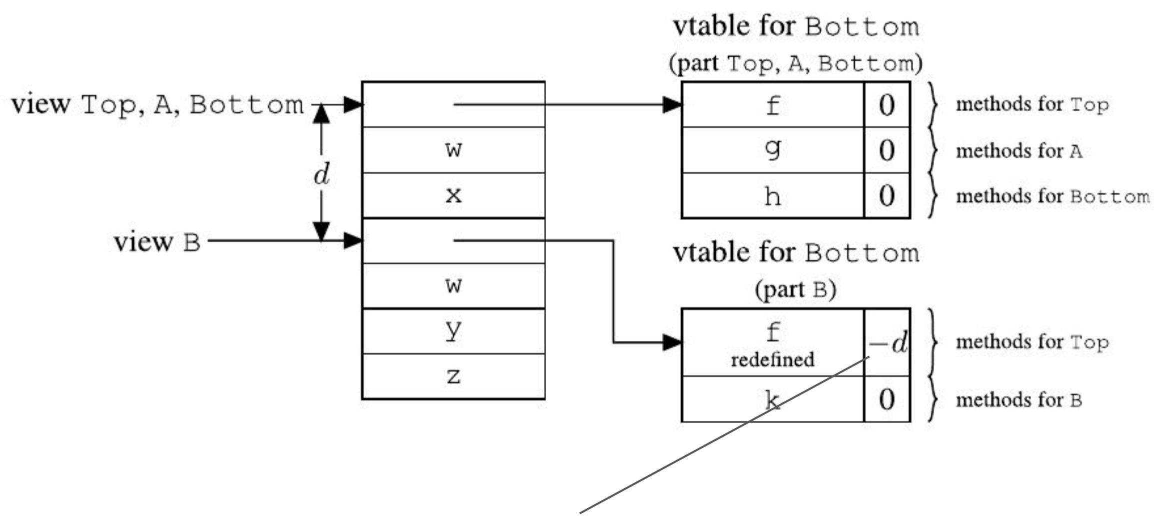


V-table & multiple inheritance

```
class Top{
    int w;
    int f(){
        return w;
    }
}
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}
class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}
class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```

Note: in C++ things more complicated since classes can be virtual (shared multiple inheritance)





Offset for the "this" pointer if f is redefined

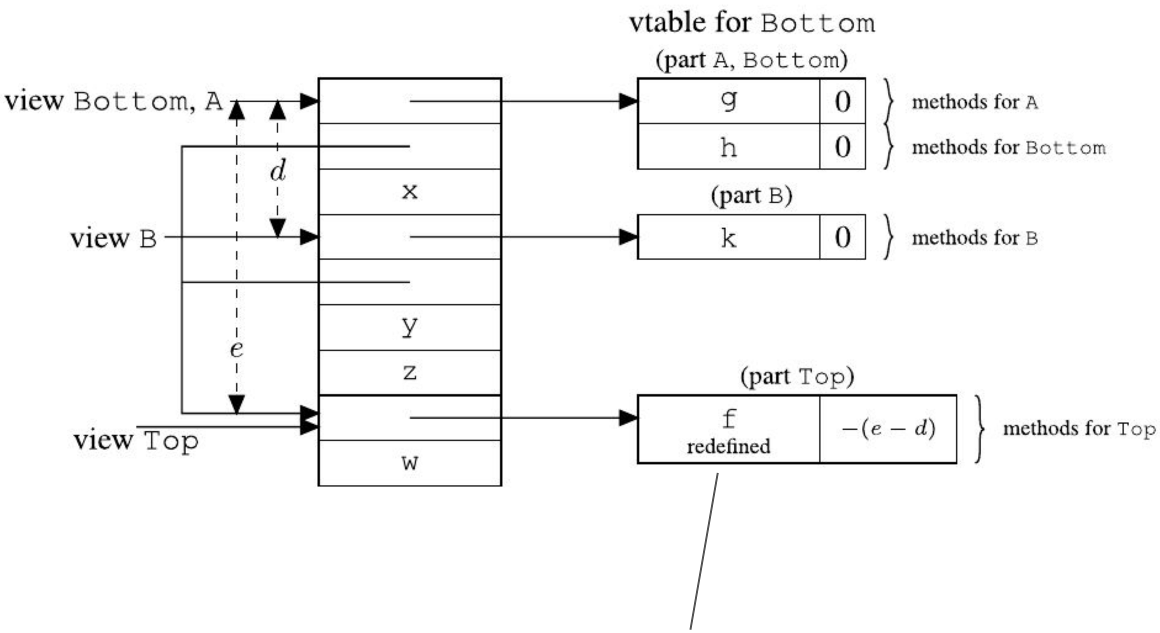
Virtual class C++

```
class A { public: void Foo() {} };

class B : public A {};

class C : public A {};

class D : public B, public C {};


D d;

d.Foo(); // is this B's Foo() or C's Foo() ??
```

- Virtual inheritance: when inheriting your classes you only want a single instance of A

Shared multiple inheritance

```
class Top{
    int w;
    int f(){
        return w;
    }
}
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}
class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}
class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```



Only one Top instance

The fragile base class problem

```
class Upper{
    int x;
    int f(){..}
}
class Lower extending Upper{
    int y;
    int g(){return y + f();}
}
```

the super class is modified as:

```
class Upper{
    int x;
    int h(){return x;}
    int f(){..}
}
```

What happens to the Lower class?

The fragile base class problem

- Changing the superclass + program compiled → subclasses no longer work correctly
  - ⇒ The offset for the method f has changed
- All subclasses should be rebuilt
  - ⇒ Non-trivial solution
- Other solution: Dynamically calculating method offsets in the vtable
  - ⇒ Problem: Efficiency

Object-oriented languages of various types

- Class-based
  - The behavior of an object determined by its class
- Object-based
  - Objects defined directly
  - Some objects serve as a template
  - The other objects are obtained *cloning* the template
  - And *delegate* to the template the execution of some methods

Object Prototypes (Delegation)

- Delegation: the principle that one object can ask another object (its parent) to execute a method for it
- JavaScript objects inherit properties and methods from a prototype (that can change at run time!)

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
var myFather = new Person("John", "Doe", 50, "blue");

Person.prototype.nationality = "English";

Person.prototype.name = function() {
    return this.firstName + " " + this.lastName;
};
```

Duck Typing

- duck test—"If it walks like a duck and it quacks like a duck, then it must be a duck"
- used to determine if an object can be used for a particular purpose
- synonym of late binding or dynamic binding:
    - the compiler does not read enough information to verify the method exists or bind its slot on the v-table
    - the method is looked up by name at runtime

Mixins

- A class that contains methods for use by other classes without having to be the parent class of those other classes
- "included" rather than "inherited"
- similar to interface with implemented methods + state
- can be composed only using the inheritance
- Languages: Simula, Flavor, Python, …
- Advantages:
  - mechanism for multiple inheritance (+ selection on only what to share)
  - code reusability

Trait

- set of methods that can be used to extend the functionality of a class
- somewhat between interface and a mixin
  - interface may define one or more behaviors via method signatures
  - trait defines full method definitions (includes the body) (e.g., new default method in Java 9)
  - mixins may carry state, traits no
- Better composition operation than mixins
  - symmetric sum (if disjoint methods) + asymmetric sum
  - exclusion (remove methods)
  - alias (add methods)

Homework

- Exercises Chapter 10 2-4