Understanding Substitution and Liskov Substitution Principle

Zoran Horvat

OWNER AT CODING HELMET CONSULTANCY

@zoranh75 www.codinghelmet.com

The Object Usability Issue

We can create an object easily

Created object may be *inconsistent*

Created object may be *incomplete*

But we cannot use an object easily

Consuming code is then in trouble

Client must work to prevent failure

Design Goals

Consumer must not be able to fail

Construction mechanism must be fail-proof

Only complete and consistent objects must come out

Covariance vs. Contravariance Mismatch

Returning a new object
is *covariant* on object type

It is easier for the consumer

It increases abstractness of types
the client depends on

Accepting objects as arguments
is *contravariant*
on argument types

Feature provider wishes to see
more concrete types as arguments

It wishes to use their concrete features

Concrete features do not exist
in abstract base types

In this module...

Substitution Principle

Defines the way we consume objects

But it may cause new kinds of troubles (*Again?!*)

Liskov Substitution Principle (LSP)

Strengthens conditions on simple Substitution Principle

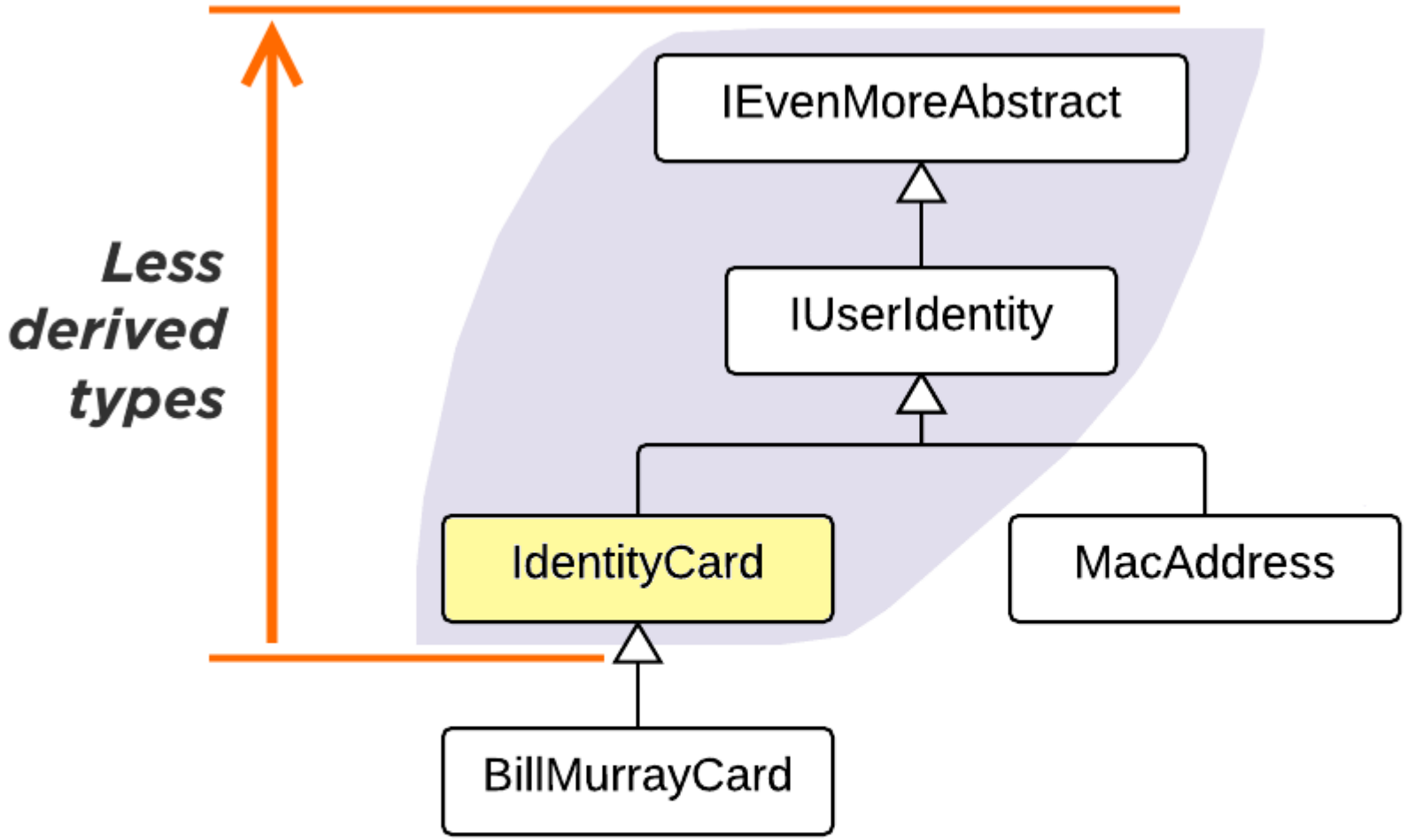
Leads to the idea of method preconditions

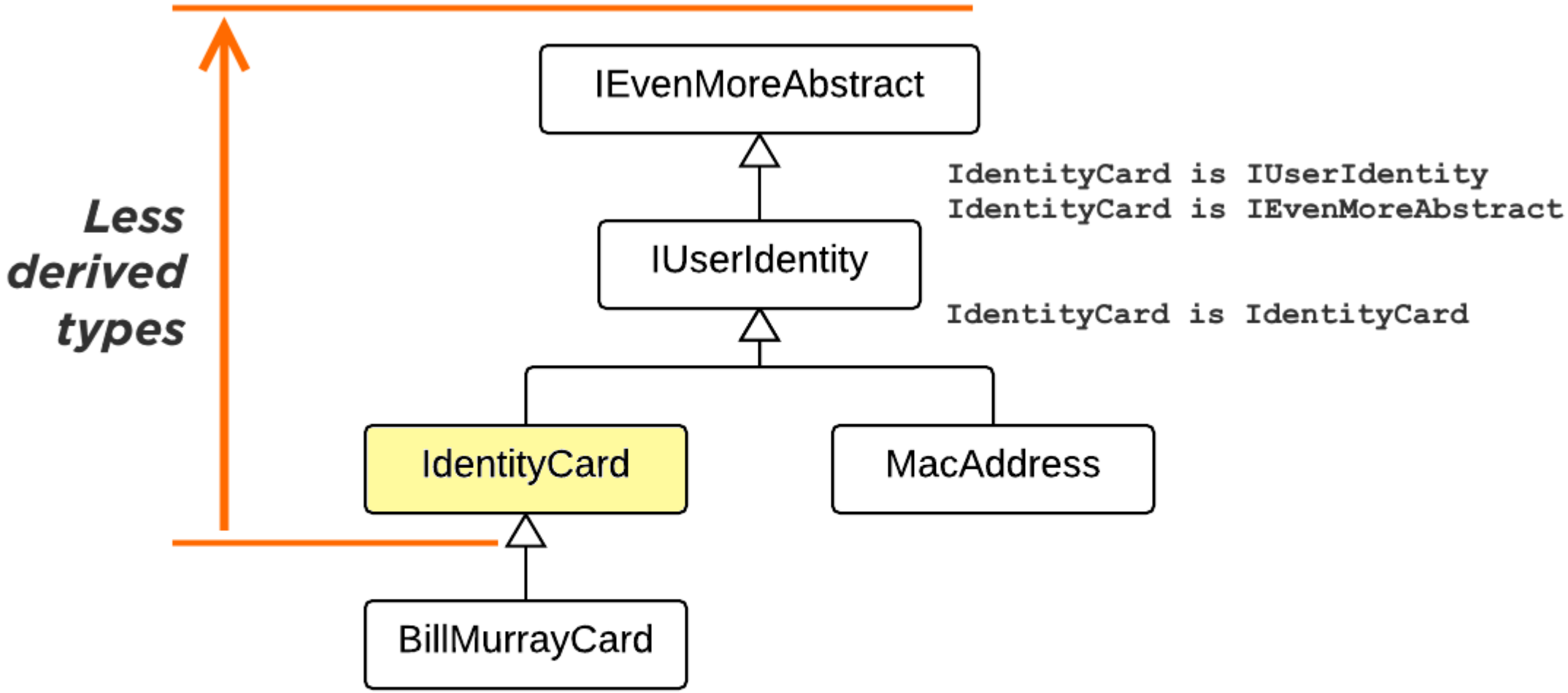
Further possibilities

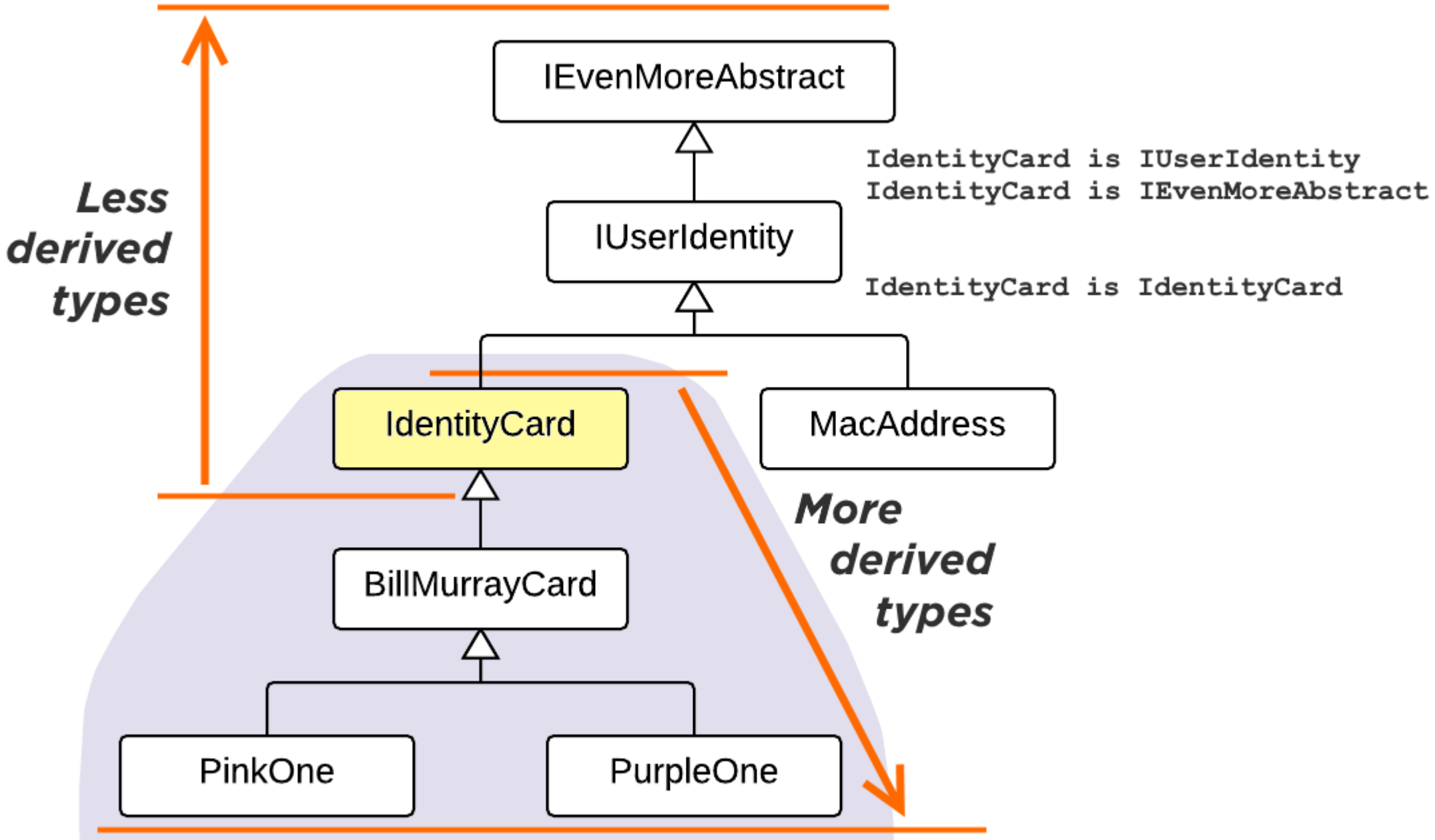
Builder and Specification pattern

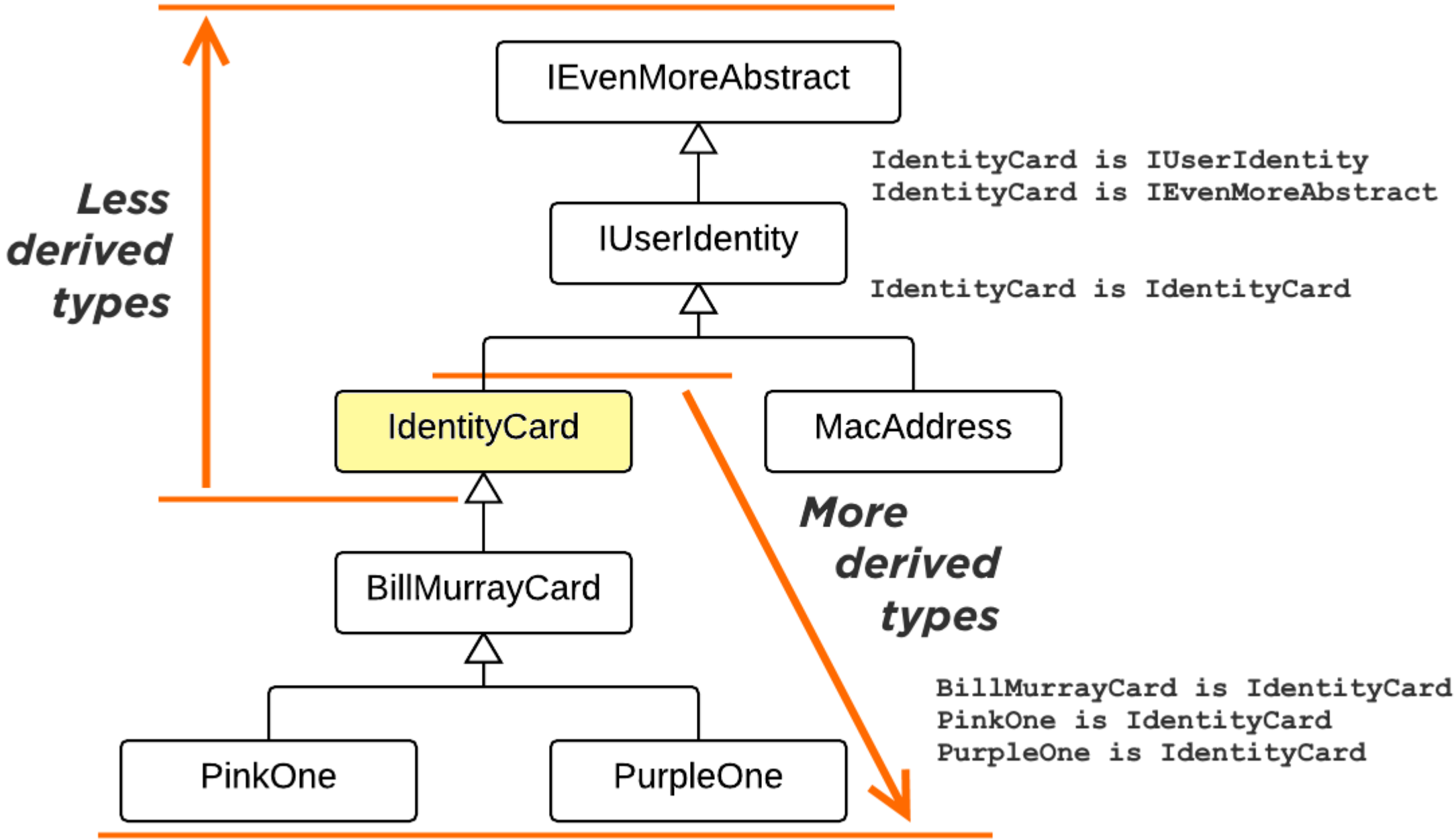
They adhere to LSP

They offer safe ways to construct objects

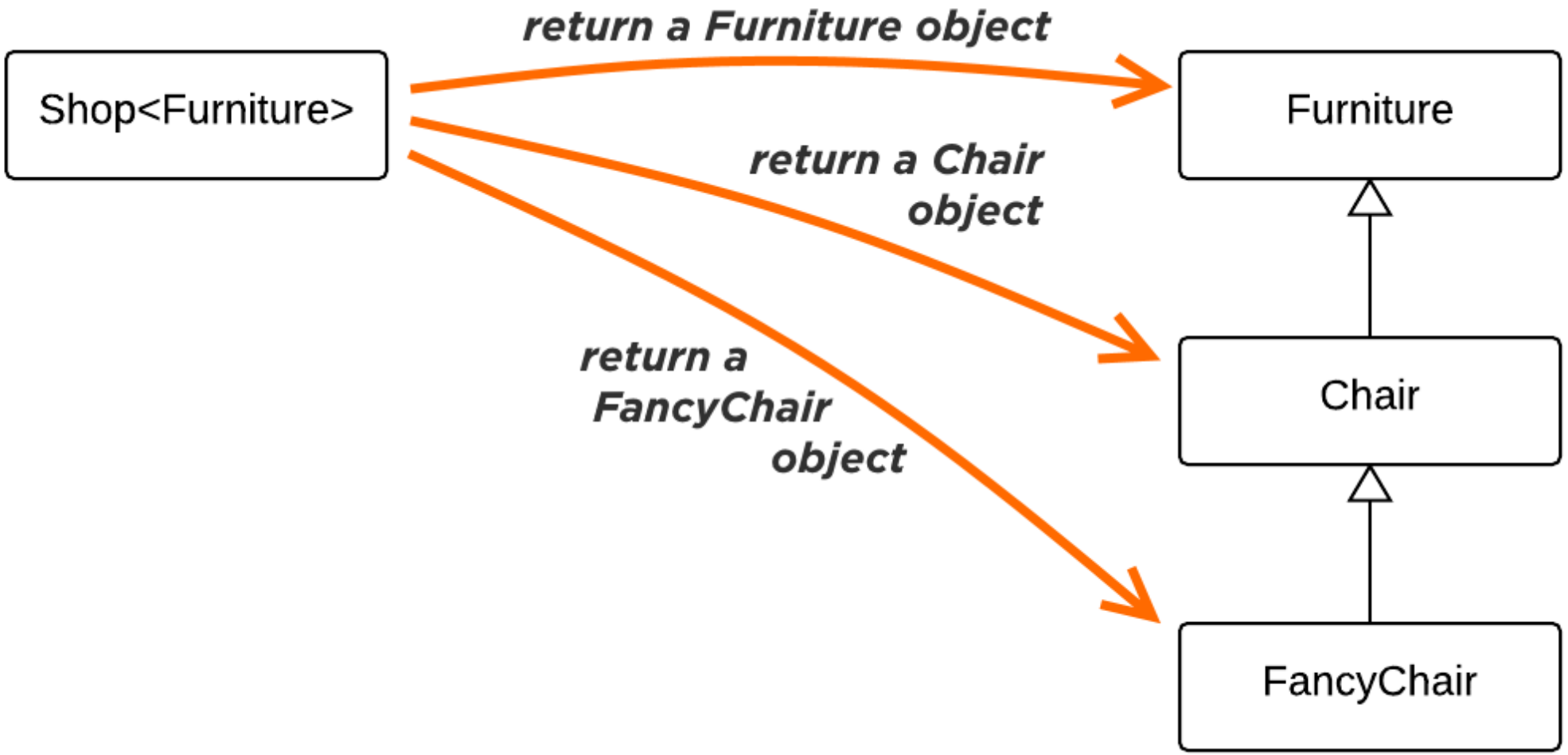

Less derived types
IEvenMoreAbstract
IUserIdentity
IdentityCard
MacAddress
BillMurrayCard


Less derived types
IEvenMoreAbstract
IUserIdentity
IdentityCard
MacAddress
BillMurrayCard
IdentityCard is IUserIdentity
IdentityCard is IEvenMoreAbstract
IdentityCard is IdentityCard

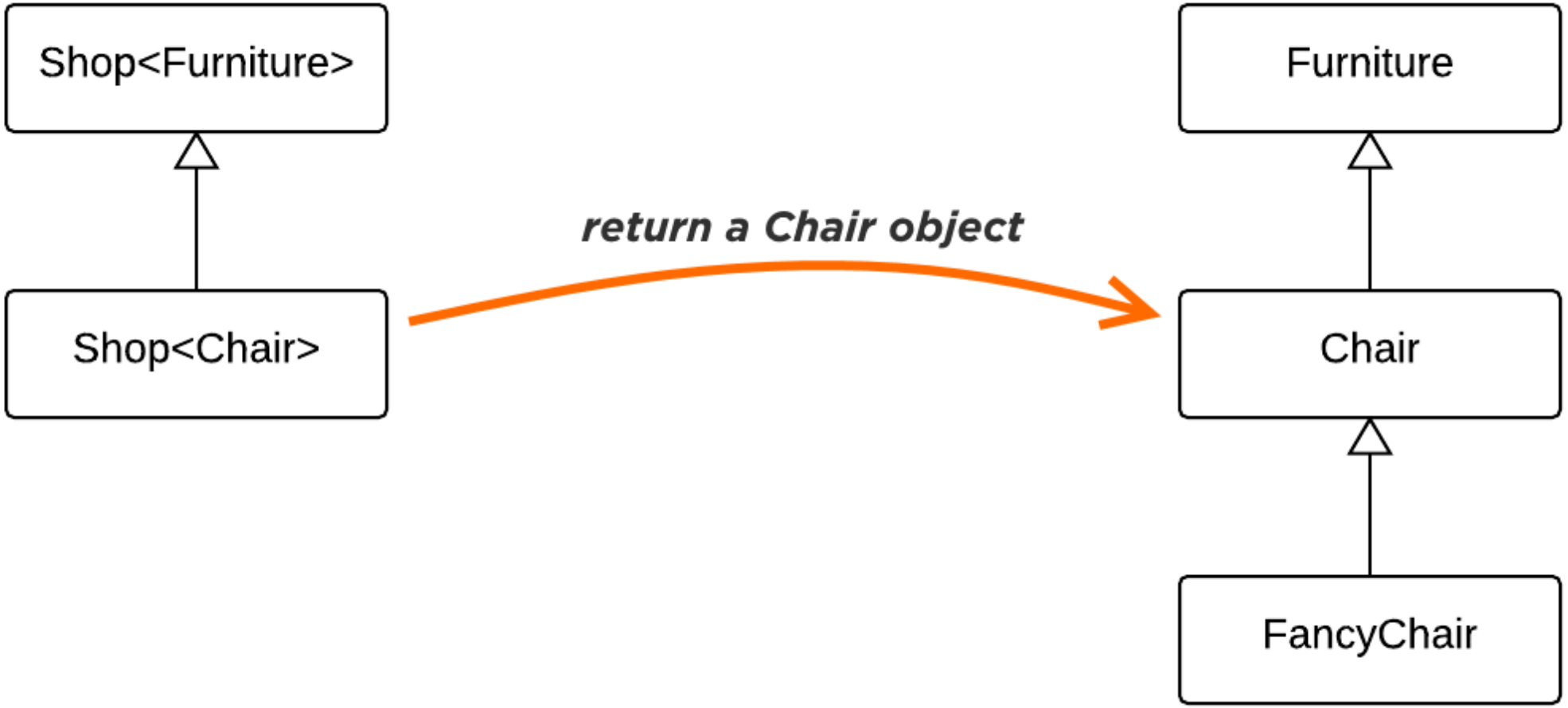

IEvenMoreAbstract
IUserIdentity
IdentityCard
MacAddress
BillMurrayCard
PinkOne
PurpleOne
Less derived types
More derived types
IdentityCard is IUserIdentity
IdentityCard is IEvenMoreAbstract
IdentityCard is IdentityCard

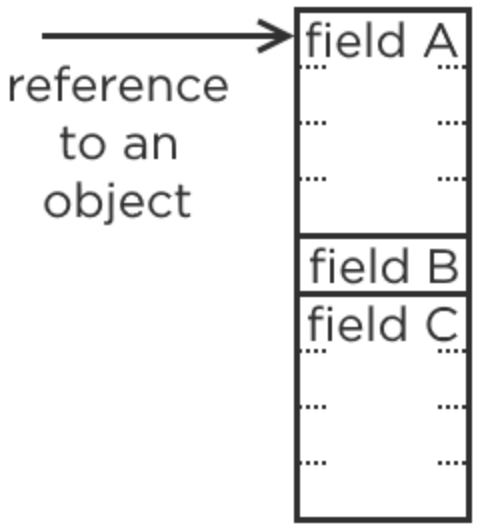

IEvenMoreAbstract
IUserIdentity
IdentityCard
MacAddress
BillMurrayCard
PinkOne
PurpleOne
Less derived types
More derived types
IdentityCard is IUserIdentity
IdentityCard is IEvenMoreAbstract
IdentityCard is IdentityCard
BillMurrayCard is IdentityCard
PinkOne is IdentityCard
PurpleOne is IdentityCard

Substitution as a Generalization Tool

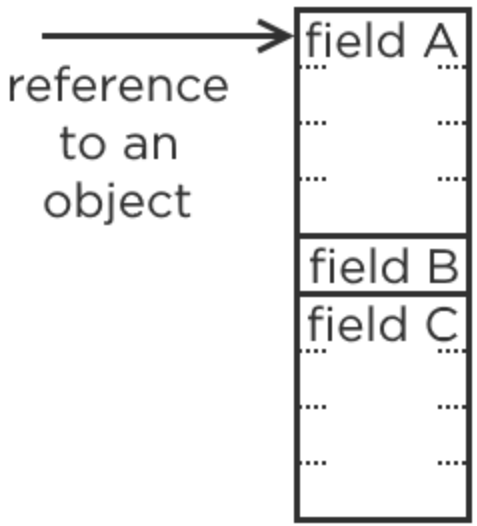


Substitution as a Generalization Tool





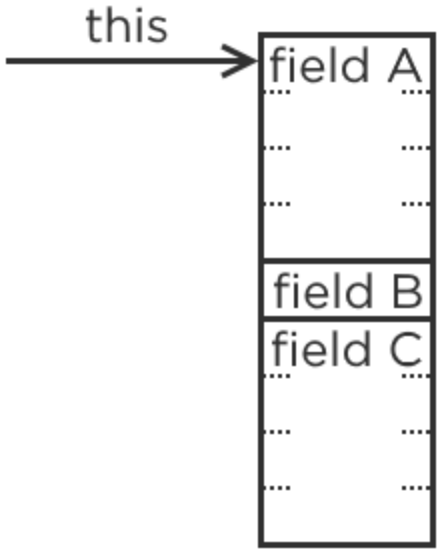

```
class Base
{
    private int fieldA;
    private byte fieldB;
    private int fieldC;
}
```




```
class Base
{
    private int fieldA;
    private byte fieldB;
    private int fieldC;

    void f()
    {
        fieldA = fieldB + fieldC;
    }
}
```

Executable code: `[base + 0B]` ← `[base + 4B]` + `[base + 5B]`

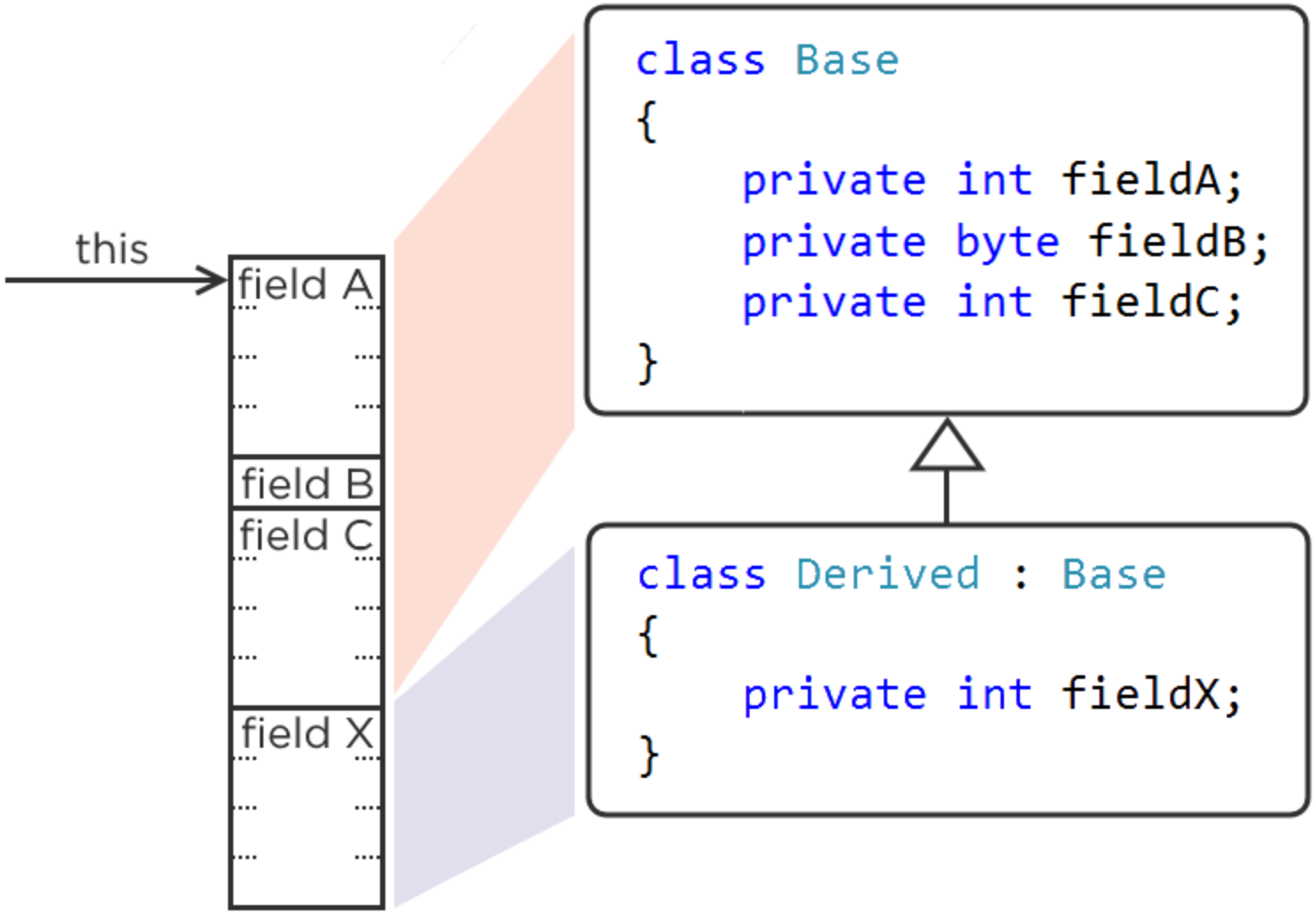


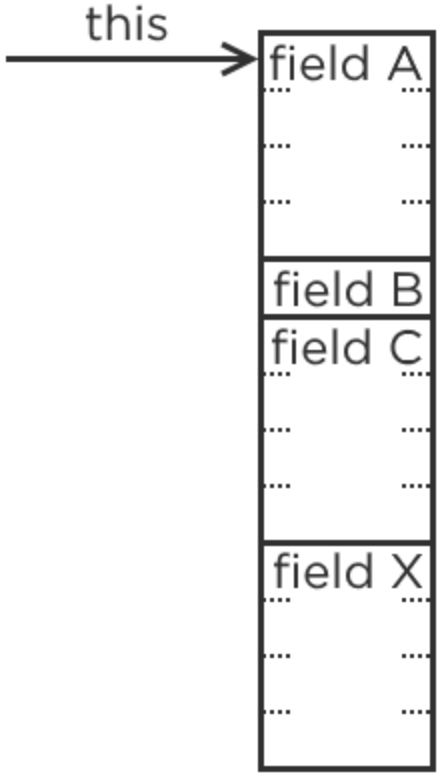

```
class Base
{
    private int fieldA;
    private byte fieldB;
    private int fieldC;

    void f(this)
    {
        this.fieldA = this.fieldB + this.fieldC;
    }
}
```

`this` reference: implicit argument to all instance-level methods

static methods have no `this` reference


this
field A
field B
field C
field X
class Base
{
private int fieldA;
private byte fieldB;
private int fieldC;
}
class Derived : Base
{
private int fieldX;
}

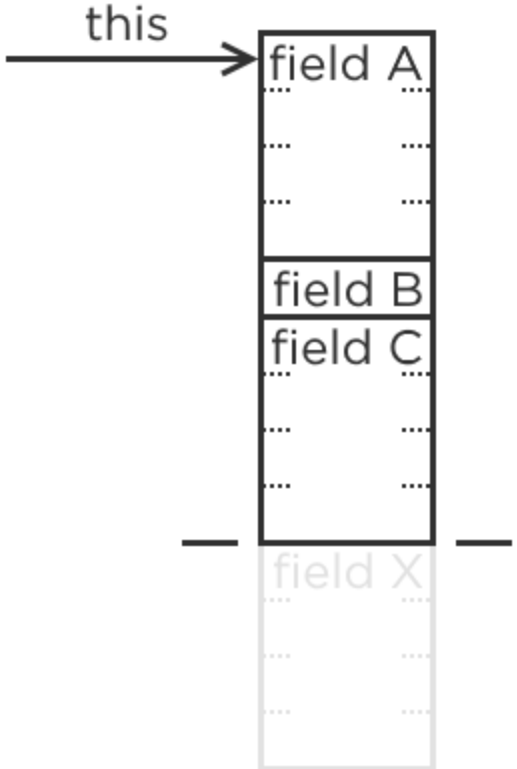



```
class Base
{
    private int fieldA;
    private byte fieldB;
    private int fieldC;
}
```

```
class Derived : Base
{
    private int fieldX;
}
```

`[base + 0B]` ← `[base + 4B]` + `[base + 5B]`

Valid in both classes

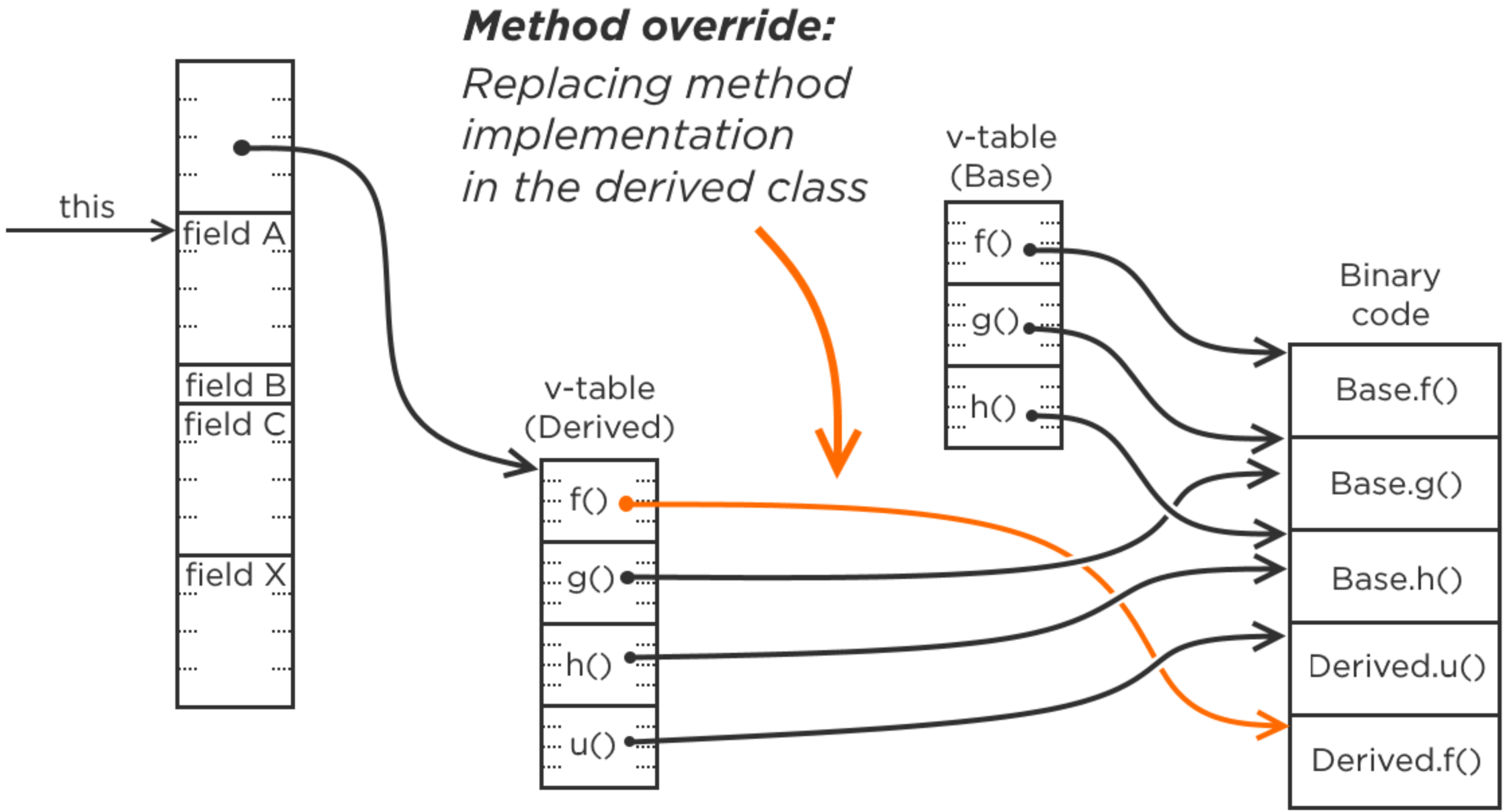


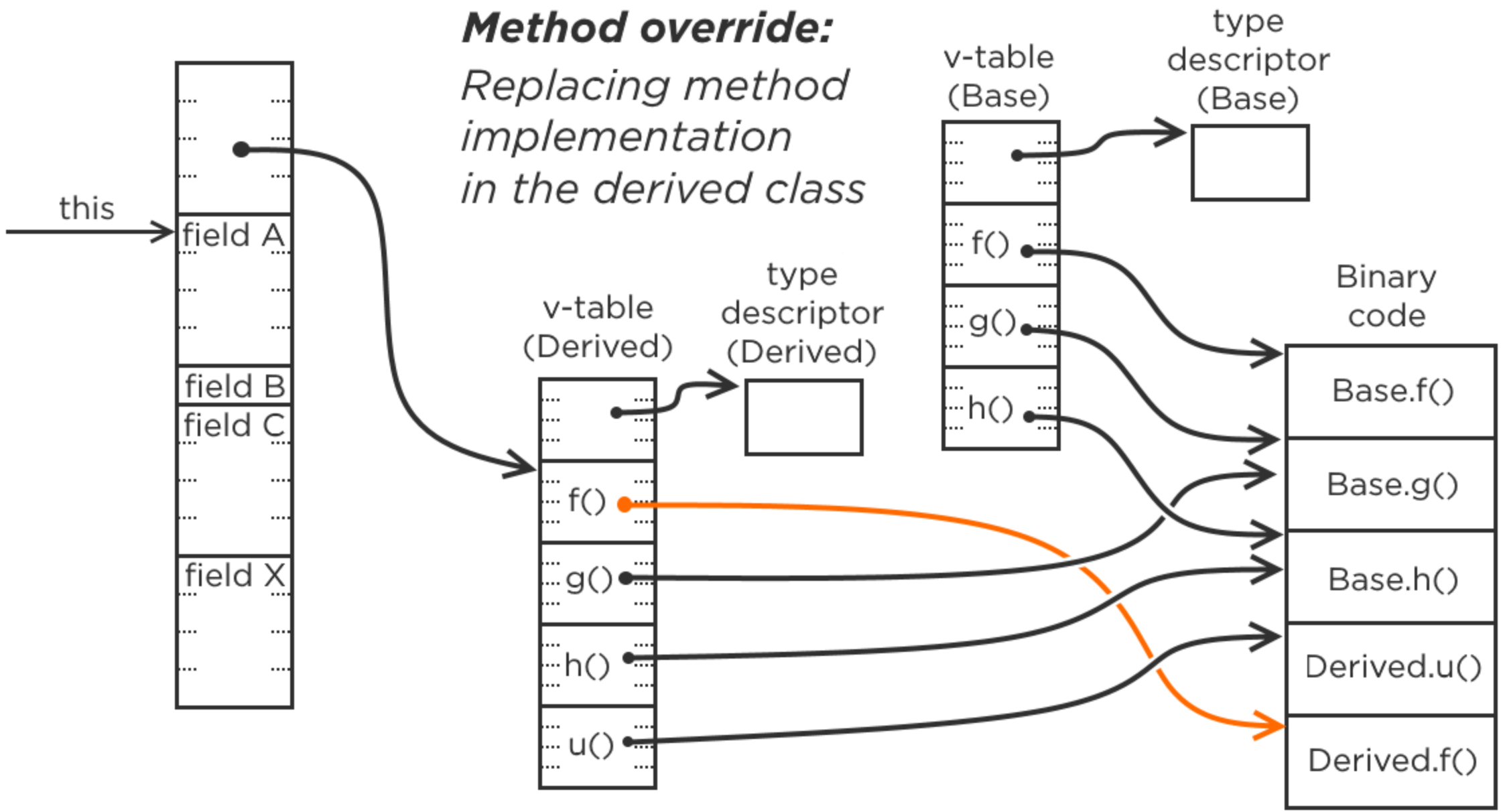

```
class Base
{
    private int fieldA;
    private byte fieldB;
    private int fieldC;
}
```

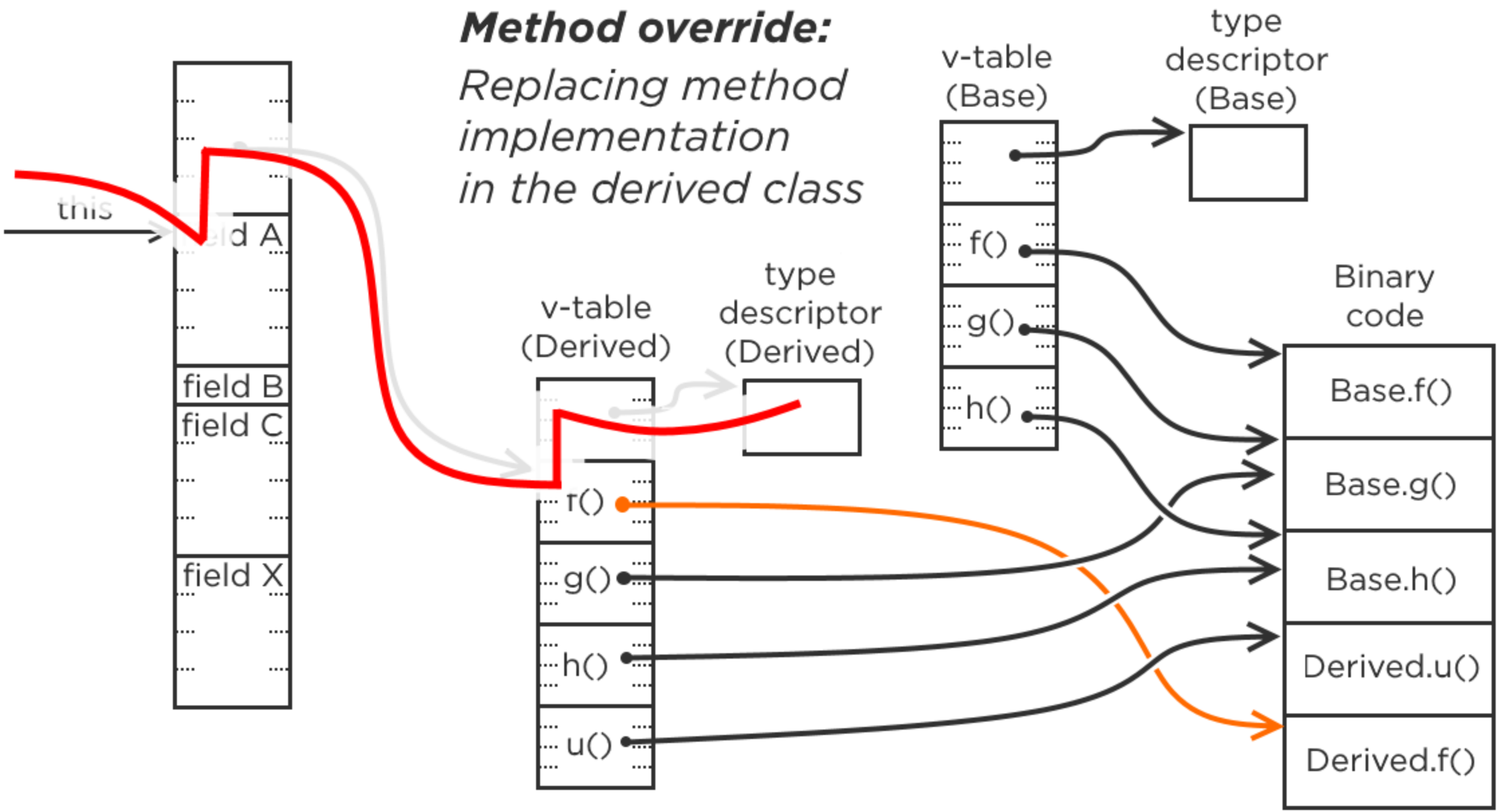
```
class Derived : Base
{
    private int fieldX;
}
```

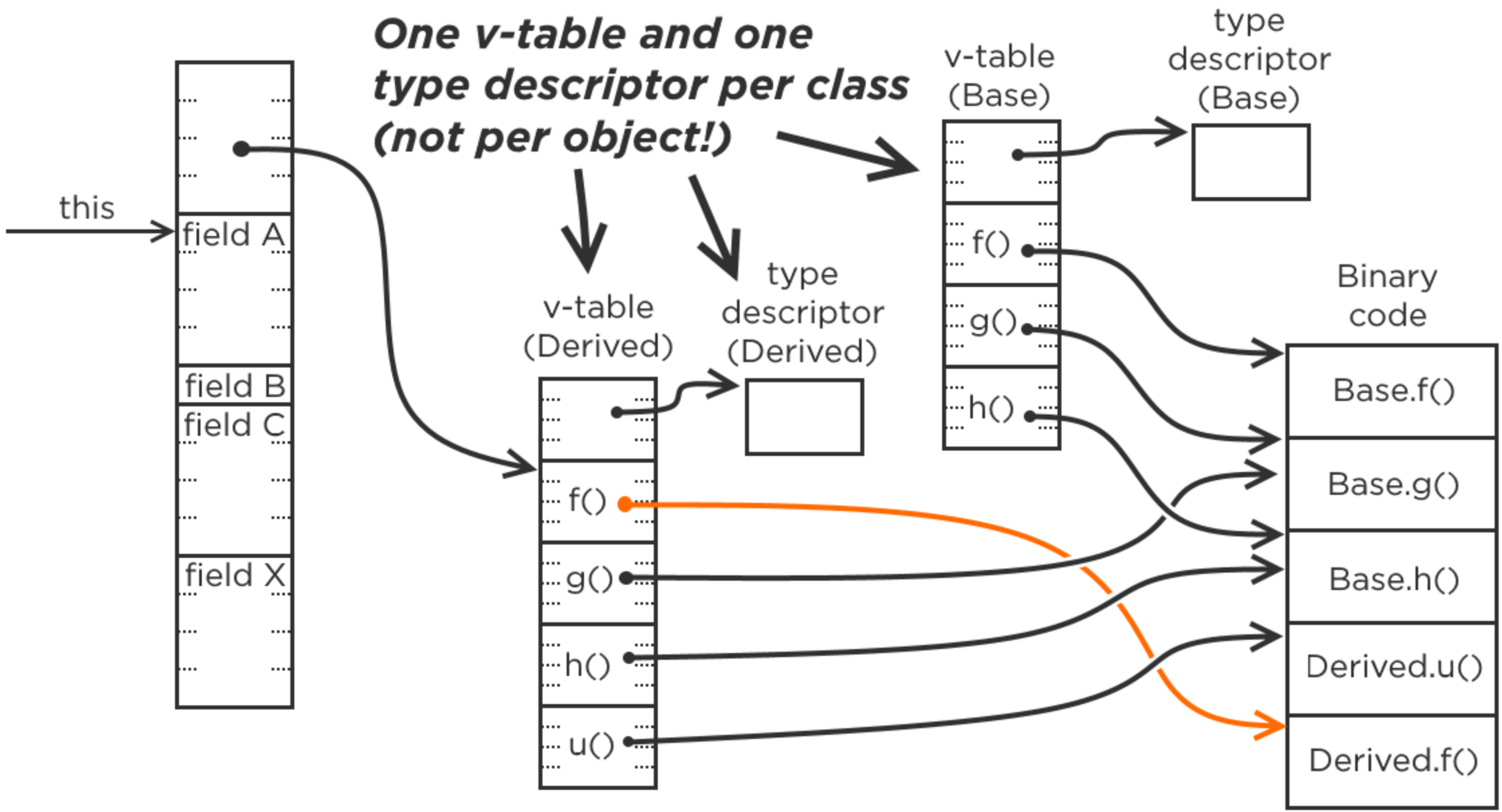
```
Base obj = new Derived();
obj.f();
```

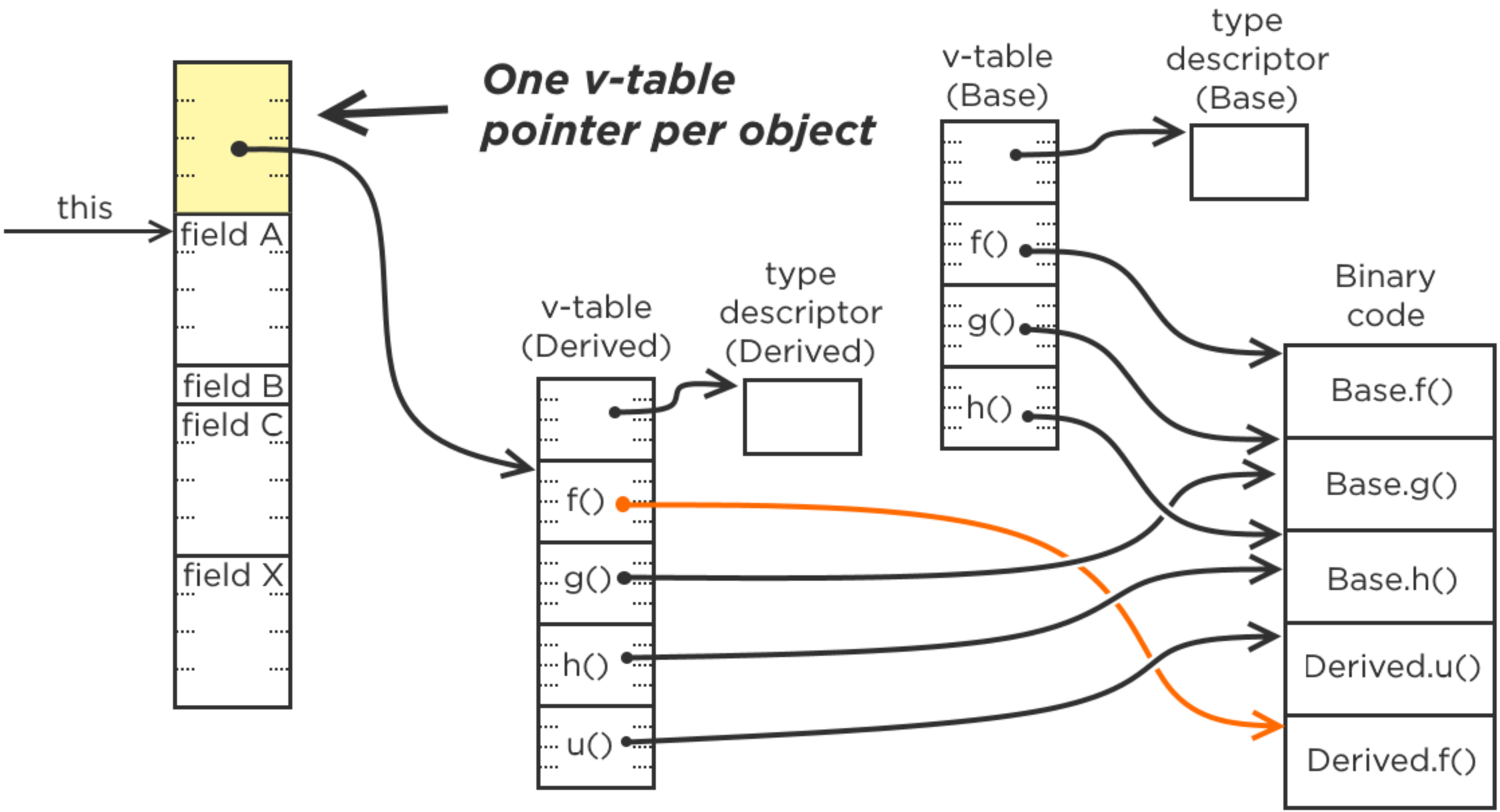
The method will keep working fine

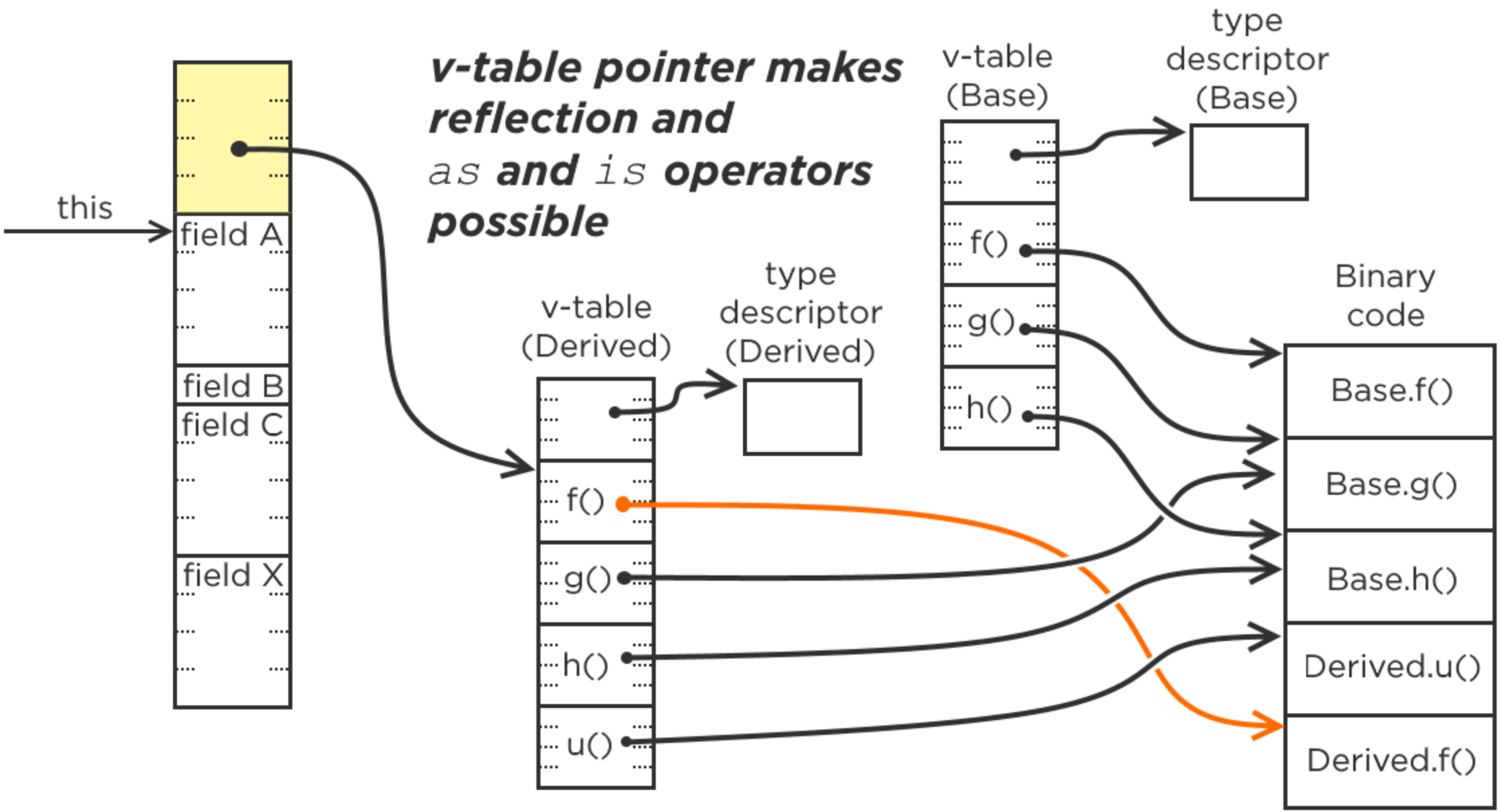

Method override:
Replacing method implementation in the derived class
this
field A
field B
field C
field X
v-table (Base)
f()
g()
h()
v-table (Derived)
f()
g()
h()
u()
Binary code
Base.f()
Base.g()
Base.h()
Derived.u()
Derived.f()

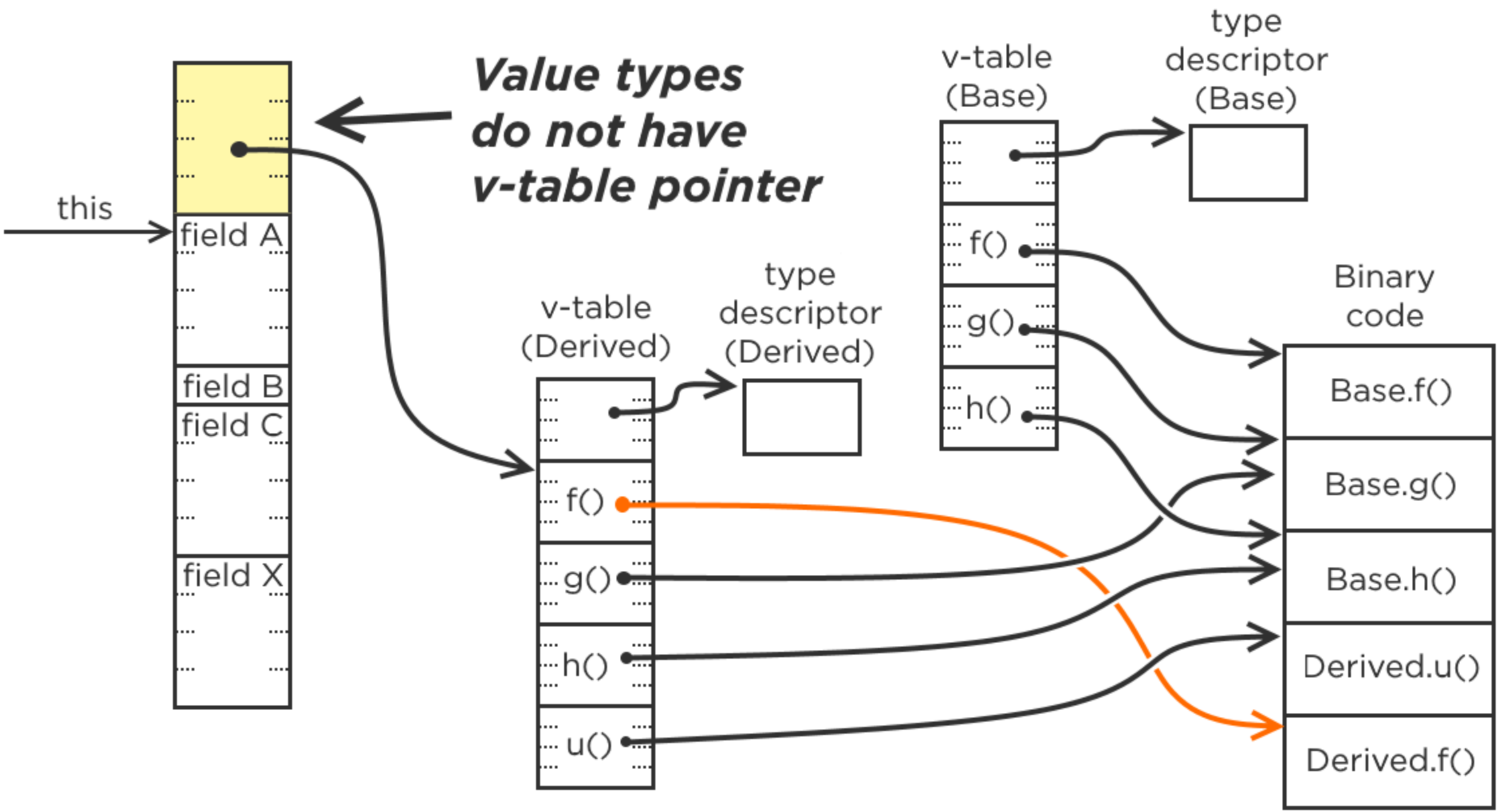

Method override:
Replacing method implementation in the derived class
this
field A
field B
field C
field X
v-table (Base)
type descriptor (Base)
f()
g()
h()
v-table (Derived)
type descriptor (Derived)
f()
g()
h()
u()
Binary code
Base.f()
Base.g()
Base.h()
Derived.u()
Derived.f()


Method override:
Replacing method implementation in the derived class
this
field A
field B
field C
field X
v-table (Base)
type descriptor (Base)
f()
g()
h()
v-table (Derived)
type descriptor (Derived)
f()
g()
h()
u()
Binary code
Base.f()
Base.g()
Base.h()
Derived.u()
Derived.f()


One v-table and one type descriptor per class (not per object!)
this
field A
field B
field C
field X
v-table (Base)
type descriptor (Base)
f()
g()
h()
v-table (Derived)
type descriptor (Derived)
f()
g()
h()
u()
Binary code
Base.f()
Base.g()
Base.h()
Derived.u()
Derived.f()


One v-table pointer per object
this
field A
field B
field C
field X
v-table (Base)
type descriptor (Base)
f()
g()
h()
v-table (Derived)
type descriptor (Derived)
f()
g()
h()
u()
Binary code
Base.f()
Base.g()
Base.h()
Derived.u()
Derived.f()


v-table pointer makes reflection and as and is operators possible
this
field A
field B
field C
field X
v-table (Derived)
type descriptor (Derived)
f()
g()
h()
u()
v-table (Base)
type descriptor (Base)
f()
g()
h()
Binary code
Base.f()
Base.g()
Base.h()
Derived.u()
Derived.f()


Value types
do not have
v-table pointer
this
field A
field B
field C
field X
v-table
(Base)
type
descriptor
(Base)
f()
g()
h()
v-table
(Derived)
type
descriptor
(Derived)
f()
g()
h()
u()
Binary
code
Base.f()
Base.g()
Base.h()
Derived.u()
Derived.f()

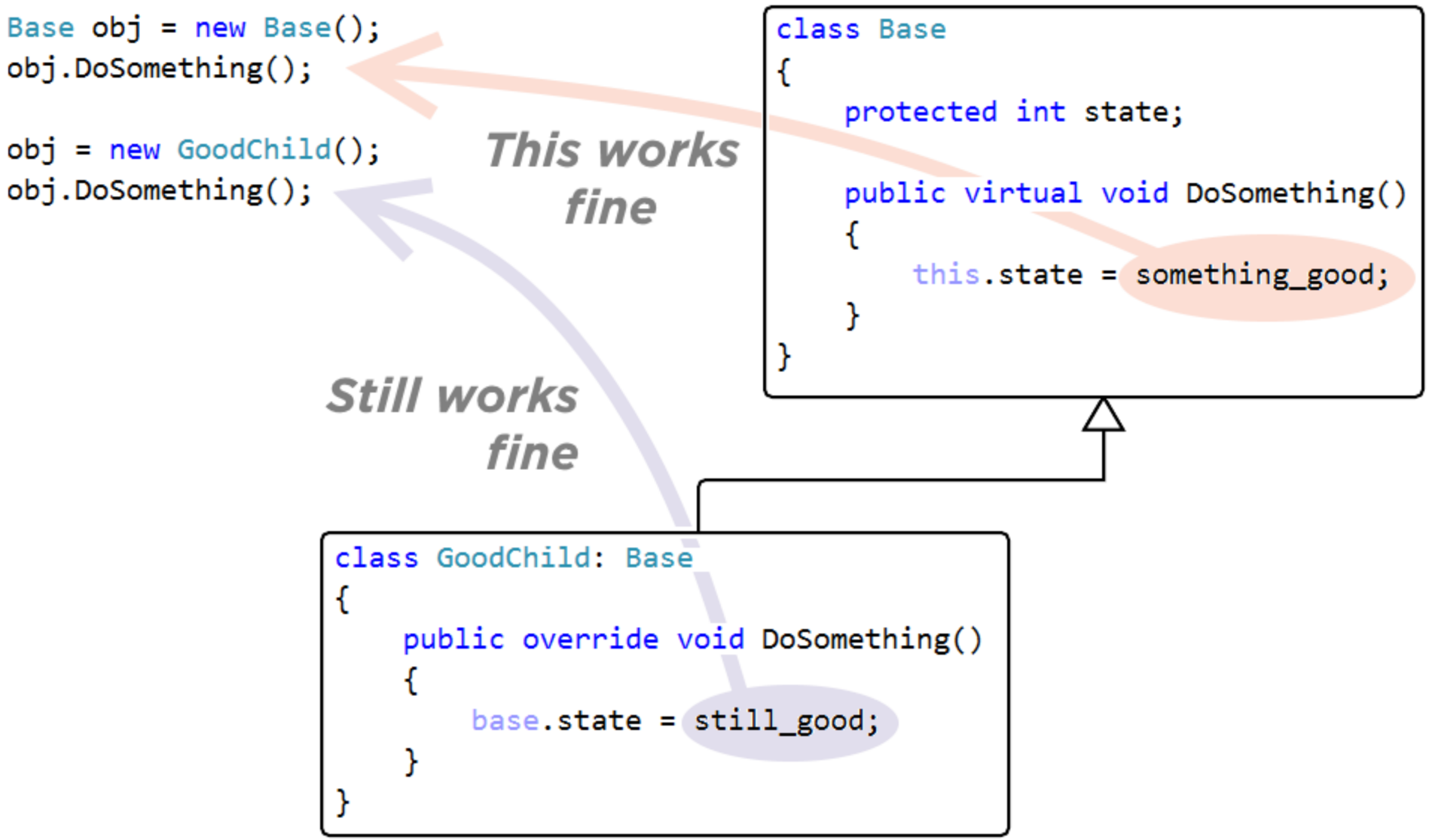
Substitution of Abstract Product

```
interface IUserFactory
{
    IUser CreateUser(string name1, string name2);
}

IUserFactory factory = new PersonFactory();

IUser user = factory.CreateUser("Max", "Planck");

return new Person("Max", "Planck");
```


Base obj = new Base();
obj.DoSomething();
obj = new GoodChild();
obj.DoSomething();
This works fine
Still works fine
class Base
{
protected int state;
public virtual void DoSomething()
{
this.state = something_good;
}
}
class GoodChild: Base
{
public override void DoSomething()
{
base.state = still_good;
}
}

```
Base obj = new Base();
obj.DoSomething();

obj = new GoodChild();
obj.DoSomething();

obj = new EvilChild();
obj.DoSomething();
```

And now we're doomed...

```
class Base
{
    protected int state;

    public virtual void DoSomething()
    {
        this.state = something_good;
    }
}
```

```
class GoodChild: Base
{
    public override void DoSomething()
    {
        base.state = still_good;
    }
}
```

```
class EvilChild: Base
{
    public override void DoSomething()
    {
        base.state = garbage;
    }
}
```

```
Base obj = new Base();
obj.DoSomething();

obj = new GoodChild();
obj.DoSomething();

obj = new EvilChild();
obj.DoSomething();
```

Next:
Liskov Substitution Principle

```
class Base
{
    protected int state;

    public virtual void DoSomething()
    {
        this.state = something_good;
    }
}
```

```
class GoodChild: Base
{
    public override void DoSomething()
    {
        base.state = still_good;
    }
}
```

```
class EvilChild: Base
{
    public override void DoSomething()
    {
        base.state = garbage;
    }
}
```

Substitution Rules

base type reference

base class

No Need to

the same behavior

s puts an emphasis on difference
between behavior and implementation

Behavior vs. Implementation

Behavior specifies ***what*** the function is doing

Implementation specifies ***how*** the function is performing its task

Behavior vs. Implementation

```
interface IUser
{
    void SetIdentity(IUserIdentity id);
}
```

```
public class Person: IUser
{
    public void SetIdentity(IUserIdentity id)
    {
        artefacts.Add(id);
    }
    ...
}
```


Behavior vs. Implementation

```
public interface IUser
{
    void SetIdentity(IUserIdentity identity);
}
```

Base type does not announce any exceptions from the SetIdentity method

```
public class Person: IUser
{
    ...
    public void SetIdentity(IUserIdentity identity)
    {
        IdentityCard idCard = identity as IdentityCard;

        if (idCard == null)
            throw new ArgumentException();

        Console.WriteLine("Accepted person identity card.");
        // do something with idCard.SSN

    }
}
```

Throwing an exception is altering behavior

Substitution and Code Correctness

Substitute only if behavior didn't change

Client relies on behavior

Changing the actual object will not endanger the client

Application execution will remain the same

Substituting an object
with an object
of a more derived type
does not affect
code correctness

Liskov Substitution Principle

Object of base type has certain qualities

Object of a derived type must have the same qualities

```
public class Person: IUser
{
    ...
    public void SetIdentity(IUserIdentity identity)
    {

        IdentityCard idCard = identity as IdentityCard;

        if (idCard == null)
            throw new ArgumentException();

        Console.WriteLine("Accepted person identity card.");
        // do something with idCard.SSN

    }
}
```

Method precondition

```
public class Person: IUser
{
    ...
    public void SetIdentity(IUserIdentity identity)
    {

        IdentityCard idCard = identity as IdentityCard;

        if (idCard == null)
            throw new ArgumentException();

        Console.WriteLine("Accepted person identity card.");
        // do something with idCard.SSN

    }
}
```

Preconditions must be satisifed before a method is invoked

```
public interface IUser
{
    void SetIdentity(IUserIdentity identity);
}
```

```
public class Person: IUser
{
    ...
    public void SetIdentity(IUserIdentity identity)
    {
        IdentityCard idCard = identity as IdentityCard;

        if (idCard == null)
            throw new ArgumentException();

        Console.WriteLine("Accepted person identity card.");
        // do something with idCard.SSN

    }
}
```

Derived class has added a precondition!

```
public interface IUser
{
    void SetIdentity(IUserIdentity identity);
}
```

```
public class Person: IUser
{
    ...
    public void SetIdentity(IUserIdentity identity)
    {

        IdentityCard idCard = identity as IdentityCard;

        if (idCard == null)
            throw new ArgumentException();

        Console.WriteLine("Accepted person identity card.");
        // do something with idCard.SSN

    }
}
```

Added precondition reduces declared capabilities

We start with some object which satisfies IUser interface

```
IUser user = some object;
```

At later time we make calls to object's methods

```
IUserIdentity id = factory.CreateIdentity();
user.SetIdentity(id);
```

We observe that the method is working fine

We start with some object which satisfies IUser interface

```
IUser user = some object;
```

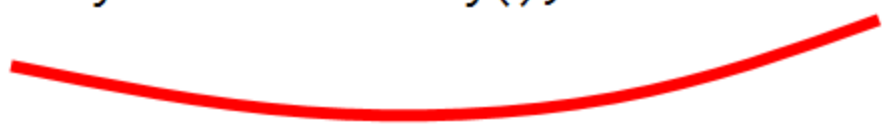
Later on, we decide to substitute the object

```
user = new Person();
```

And we repeat the same call

```
IUserIdentity id = factory.CreateIdentity();
user.SetIdentity(id);
```

But this time the call fails!



Summary

Substitution Principle

- Deals with structural subtyping
- Substitution doesn't affect *syntactical* correctness of code
- But it doesn't guarantee *semantic* correctness of code

Liskov Substitution Principle

- Deals with behavioral subtyping
- Adds rules to obey in subtypes
- Subtypes must not change behavior
- Subtypes can change implementation

Summary

Difference between behavior and implementation

- Behavior defined by public interface
- Implementation is encapsulated
- Implementation provides declared behavior

Enforcing Liskov Substitution Principle

- Preconditions added to methods
- Client ensures that preconditions hold
- Abstract type offers public interface to test preconditions

Summary

Separation of responsibilities

- Client takes recovery if preconditions don't hold
- Implementation is free to fail if preconditions don't hold

Summary

The problem of Abstract Factory

- The client may accidentally break the Liskov Substitution Principle
- Dangerous segments of code may be closed into a separate class
  - This will lead to Builder and Specification patterns

Next module –

Returning to Concrete Classes with the Builder Pattern