Returning to Concrete Classes with the Builder Pattern

Zoran Horvat

OWNER AT CODING HELMET CONSULTANCY

@zoranh75 www.codinghelmet.com

Abstract vs. Concrete Types

Key elements

Abstract types

Polymorphic execution

Object substitution

Polymorphism

Object substitution enables polymorphism

Polymorphism may be hard to control

Especially hard when there are many interacting objects

Abstract Factory

Enforces polymorphic products

Issues when products have to interact

Leads to less type safety

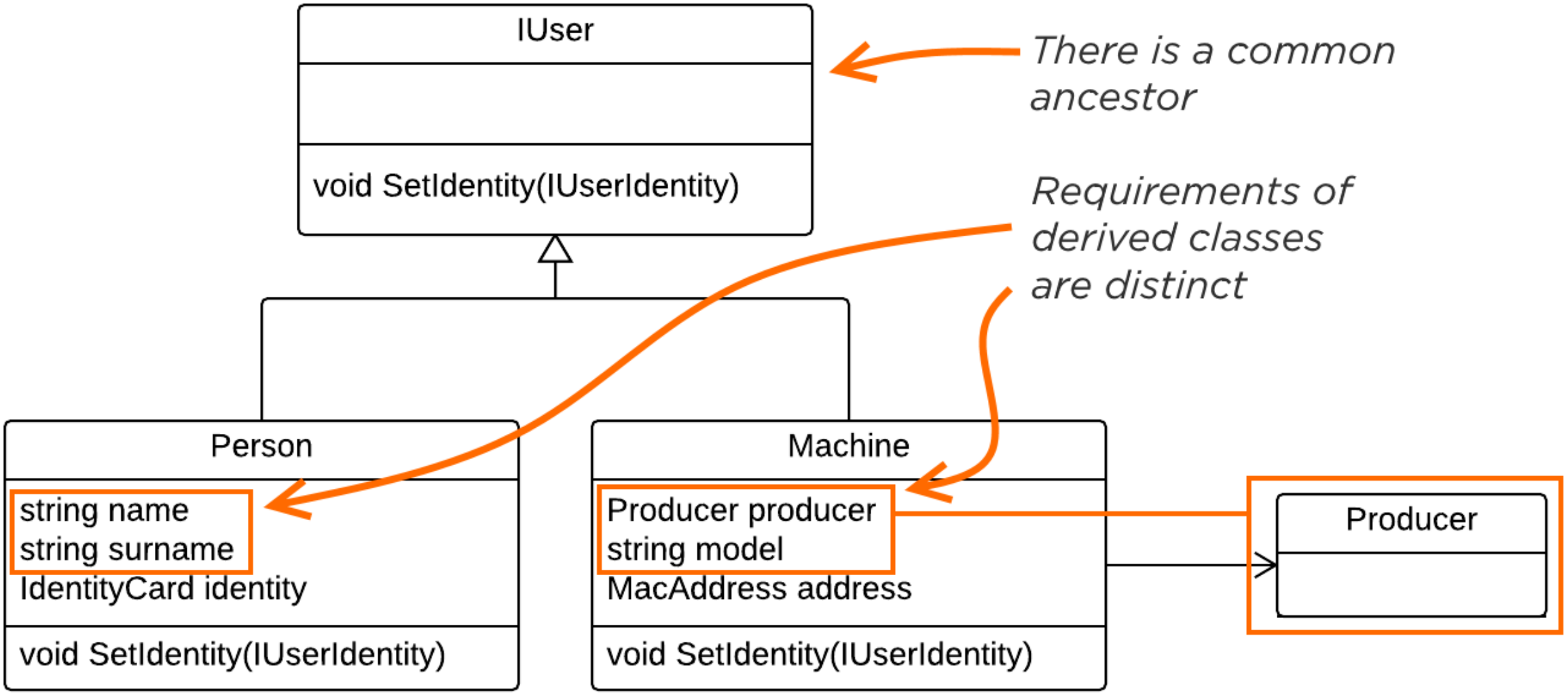
Abandoning Abstractness

Focus the factory on only one kind of products

Move concrete features up into the abstract product

Next step:
Abandon all abstractness

Focus on concrete rules to create one concrete product


IUser
void SetIdentity(IUserIdentity)
There is a common ancestor
Requirements of derived classes are distinct
Person
string name
string surname
IdentityCard identity
void SetIdentity(IUserIdentity)
Machine
Producer producer
string model
MacAddress address
void SetIdentity(IUserIdentity)
Producer

Builder Design Pattern

Deals with concrete types

Focuses on the process of constructing a new object

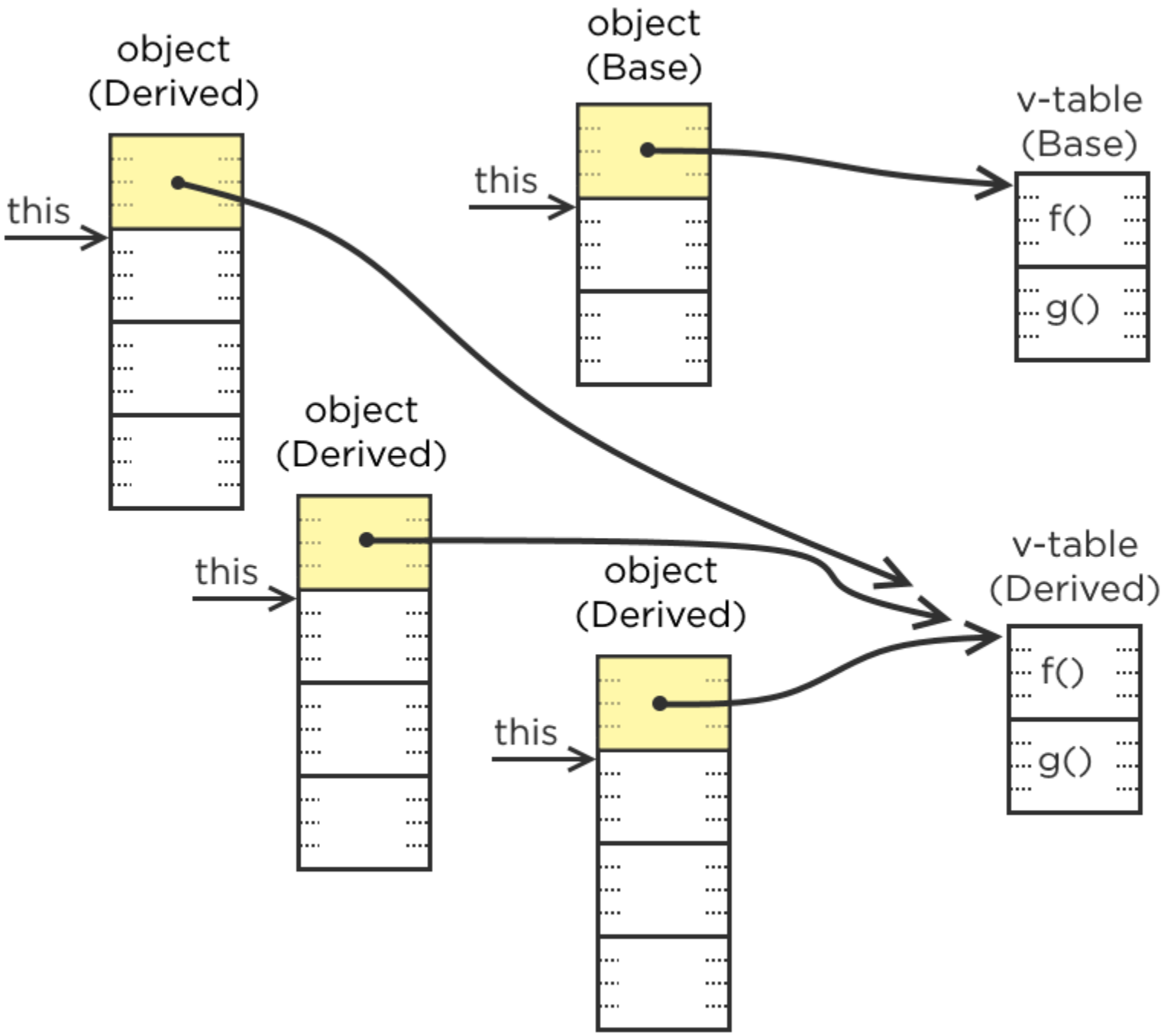
End result is a complete and consistent product object

Builder ensures consistency rules

```
class Base
{
    public virtual void f() { }
    public virtual void g() { }
}
```

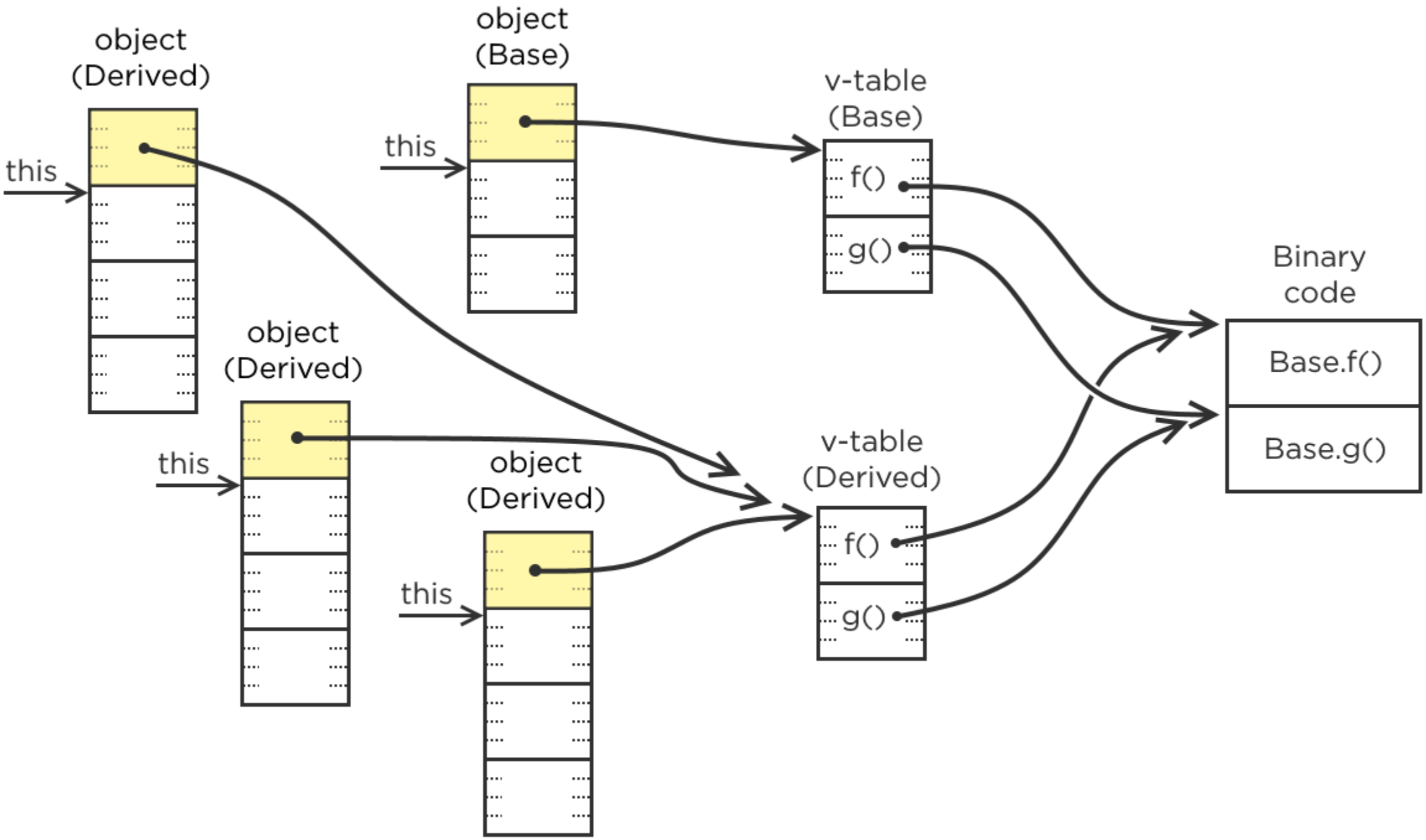
Must use `virtual` keyword in C#

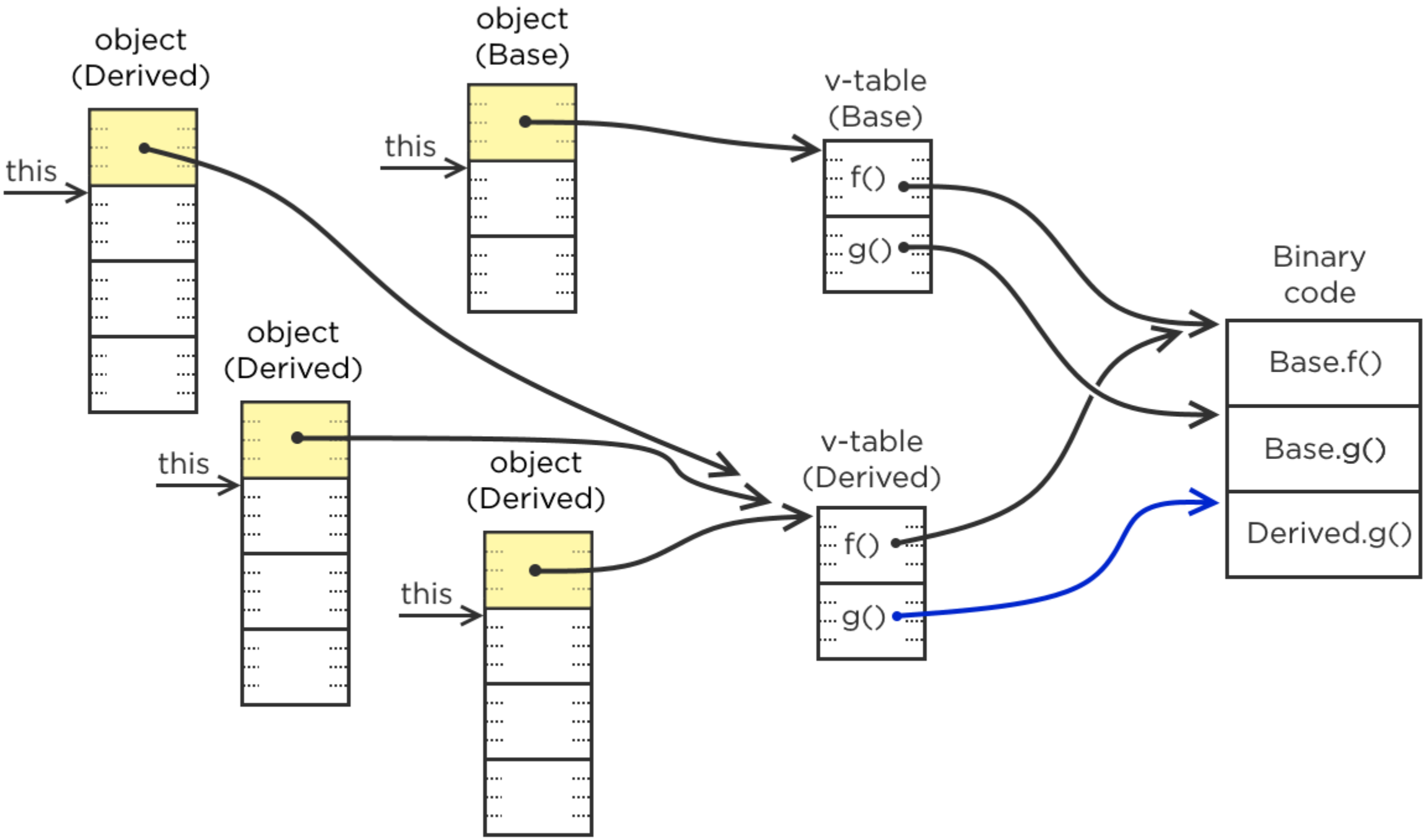
```
class Derived : Base
{
    public override void g() { }
}
```




```
class Base
{
    public virtual void f() { }
    public virtual void g() { }
}
```

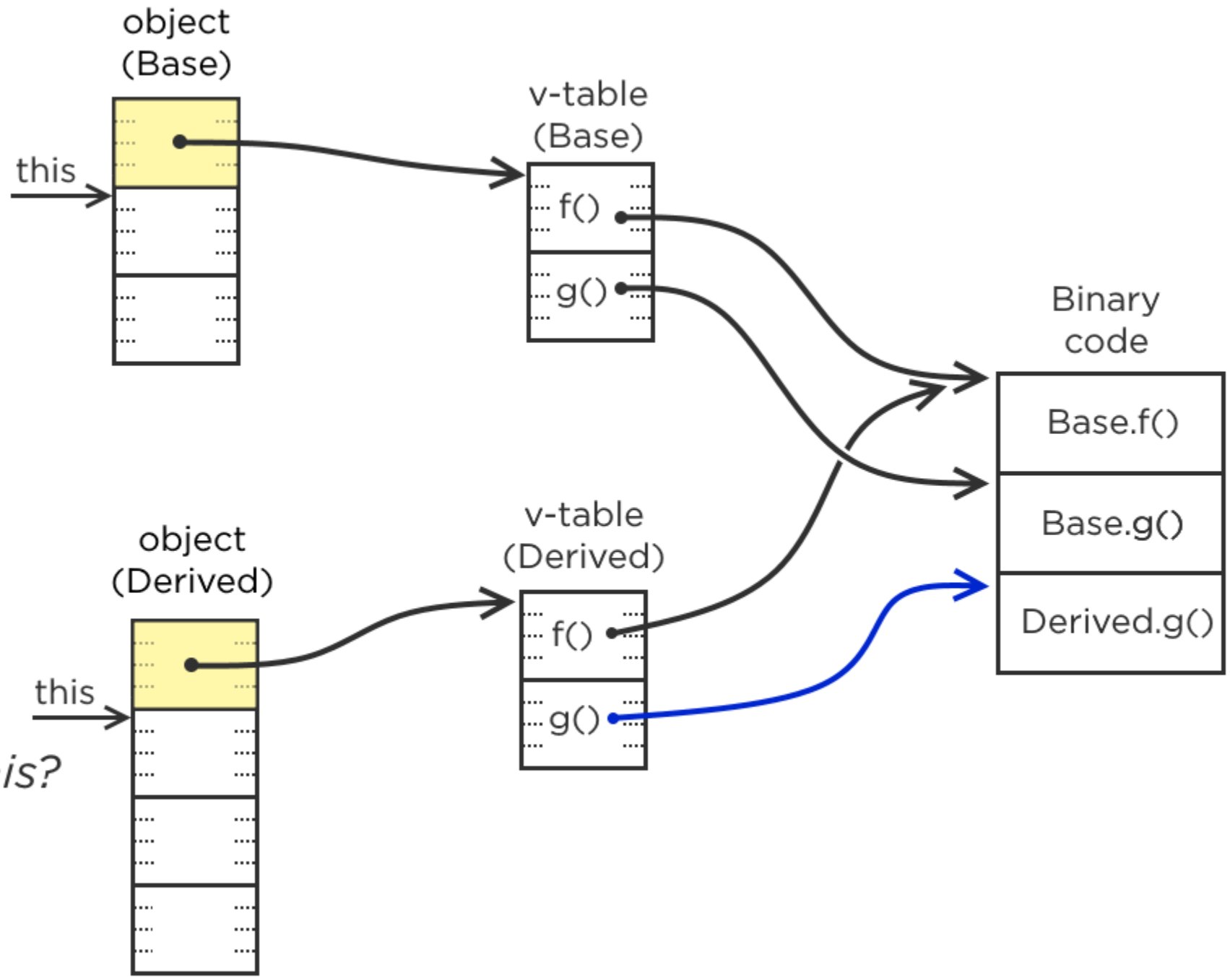
```
class Derived : Base
{
    public override void g() { }
}
```

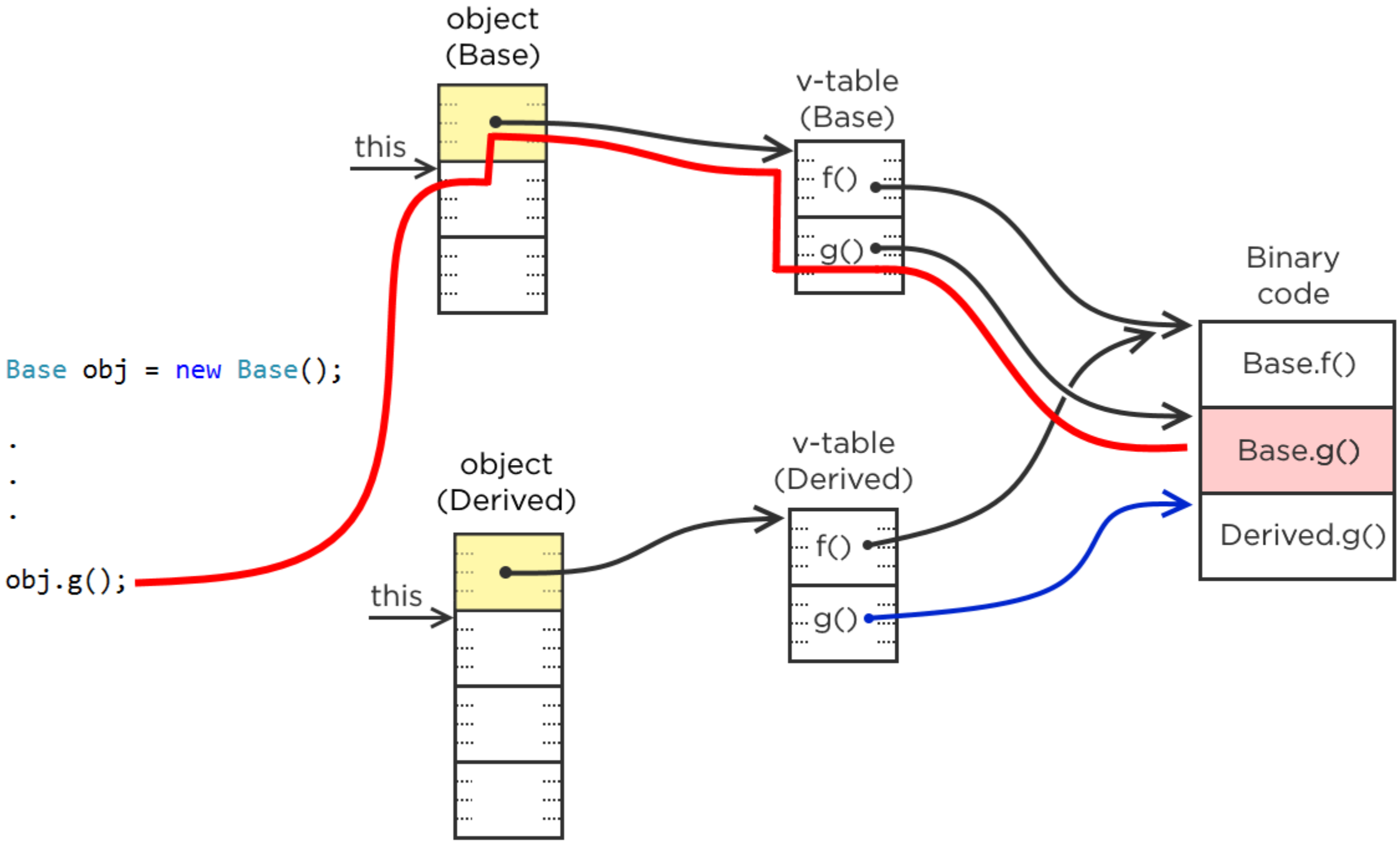

object
(Derived)
this
object
(Base)
this
v-table
(Base)
f()
g()
Binary
code
Base.f()
Base.g()
object
(Derived)
this
object
(Derived)
this
v-table
(Derived)
f()
g()

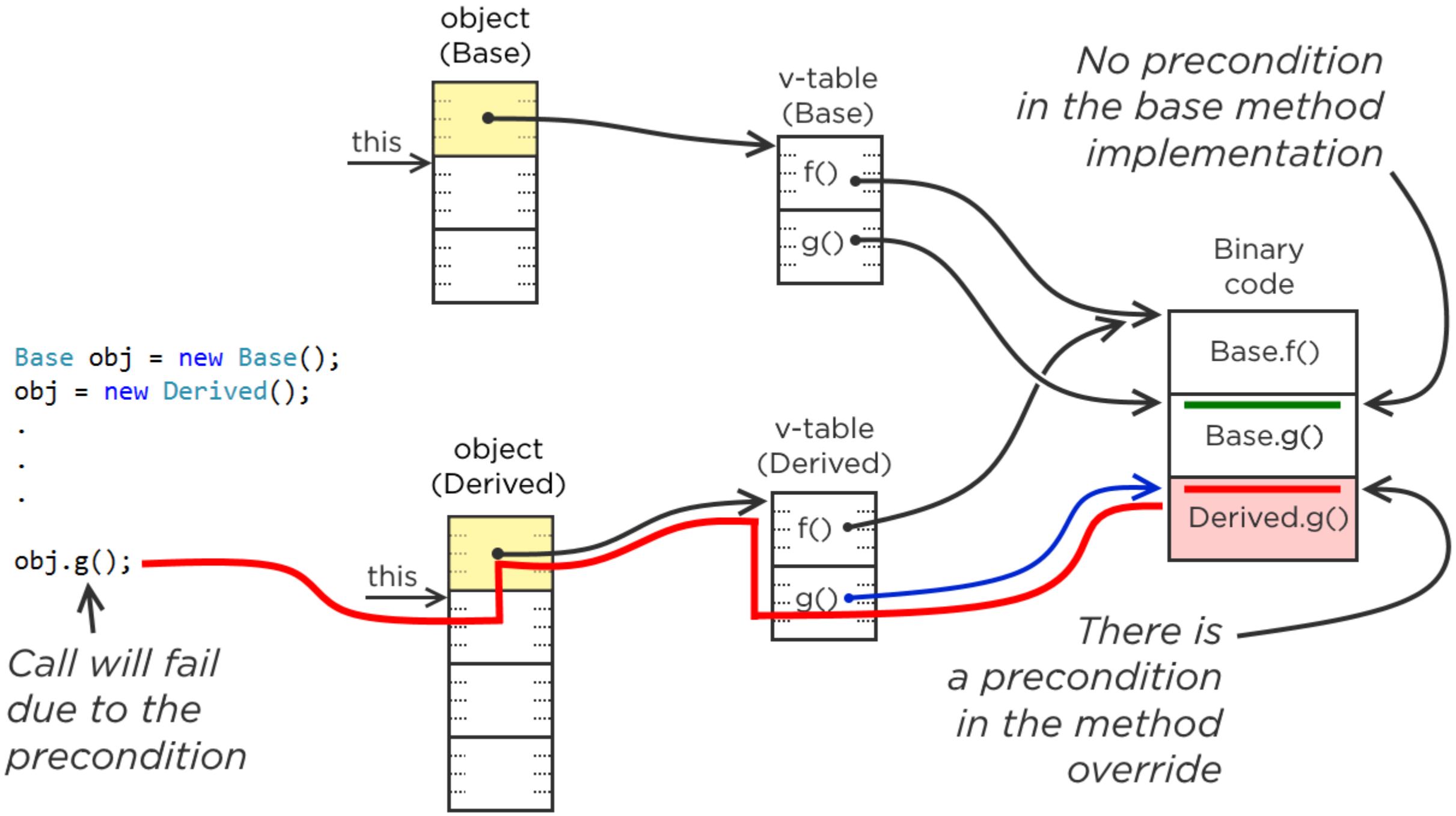

object
(Derived)
this
object
(Base)
this
v-table
(Base)
f()
g()
Binary
code
Base.f()
Base.g()
Derived.g()
object
(Derived)
this
object
(Derived)
this
v-table
(Derived)
f()
g()

```
Base obj = new Base();

.
.
.

obj.g();
```

Which object is this?
What is its type?

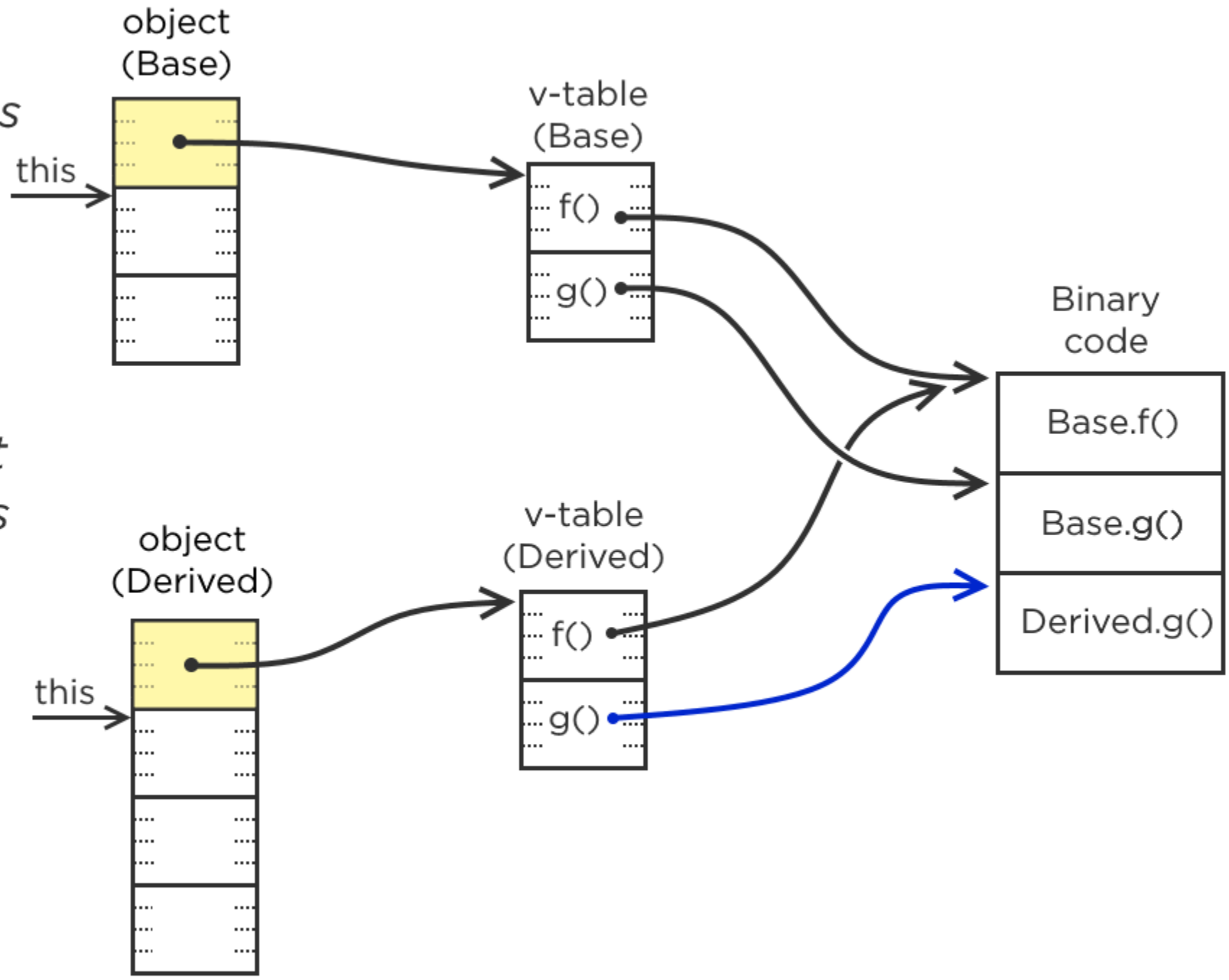


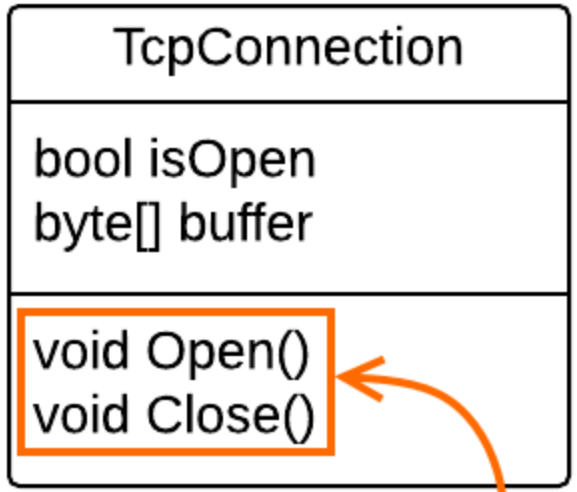

object
(Base)
this
v-table
(Base)
f()
g()
Binary
code
Base.f()
Base.g()
Derived.g()
Base obj = new Base();
.
.
.
obj.g();
object
(Derived)
this
v-table
(Derived)
f()
g()


object
(Base)
this
v-table
(Base)
f()
g()
No precondition
in the base method
implementation
Binary
code
Base.f()
Base.g()
Derived.g()
Base obj = new Base();
obj = new Derived();
.
.
.
obj.g();
object
(Derived)
this
v-table
(Derived)
f()
g()
Call will fail
due to the
precondition
There is
a precondition
in the method
override

Rules:

1. *No preconditions in method overrides*
2. *Constructor belongs to the class, not to the object*
3. *No such thing as constructor overriding*
4. *Preconditions allowed in constructors of derived classes*

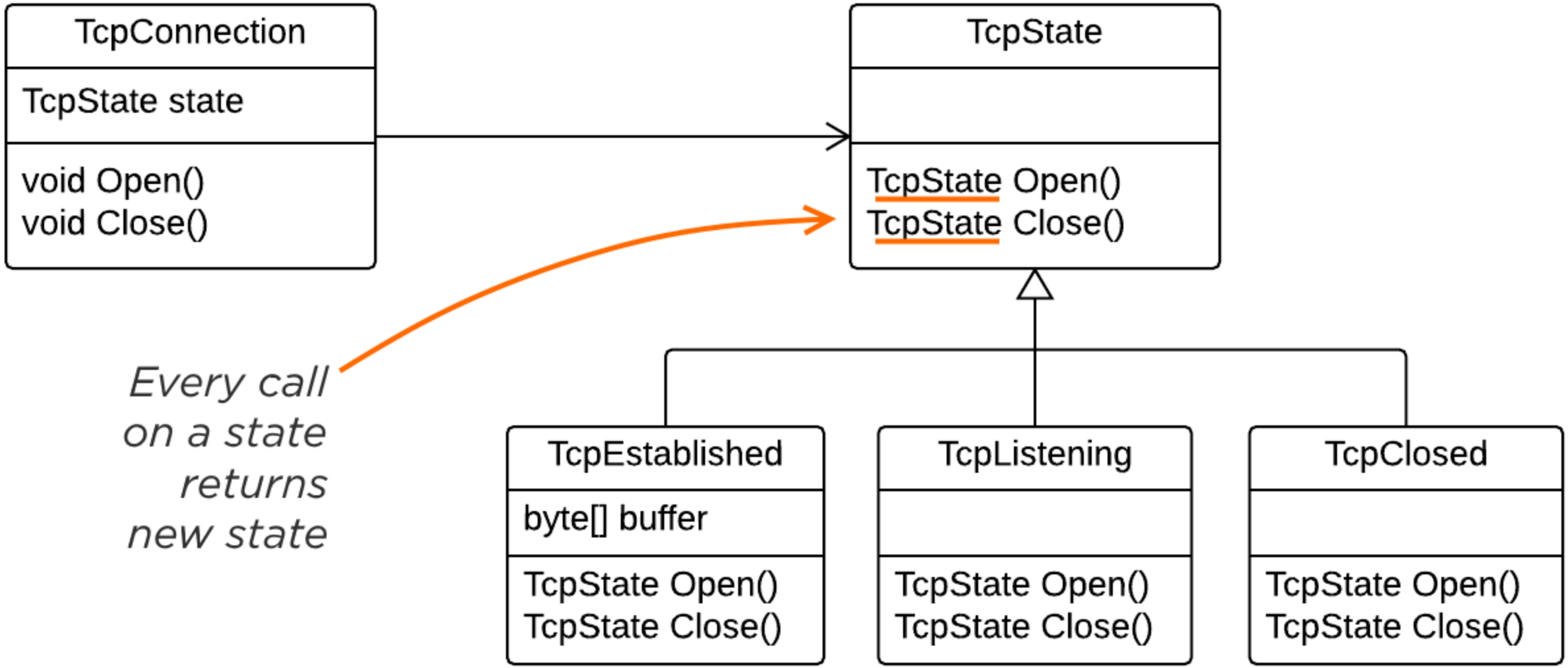




Established
Listening
Closed

Connection allows certain operations

Method implementations may become complicated


TcpConnection
TcpState state
void Open()
void Close()
TcpState
TcpState Open()
TcpState Close()
Every call on a state returns new state
TcpEstablished
byte[] buffer
TcpState Open()
TcpState Close()
TcpListening
TcpState Open()
TcpState Close()
TcpClosed
TcpState Open()
TcpState Close()

Summary

Basic implementation of the Builder

- It fails to make the client do the right things

Builder + State pattern

- Enforcing valid state transitions

Summary

Further enhancements

- Add Interface Segregation Principle to the State implementation
- Add Command-Query Separation to the Builder implementation

Ultimate design goal

- Build method will never fail if we ever reached the point where we can call it

Next module –
Embedding Calling Protocols into the Builder