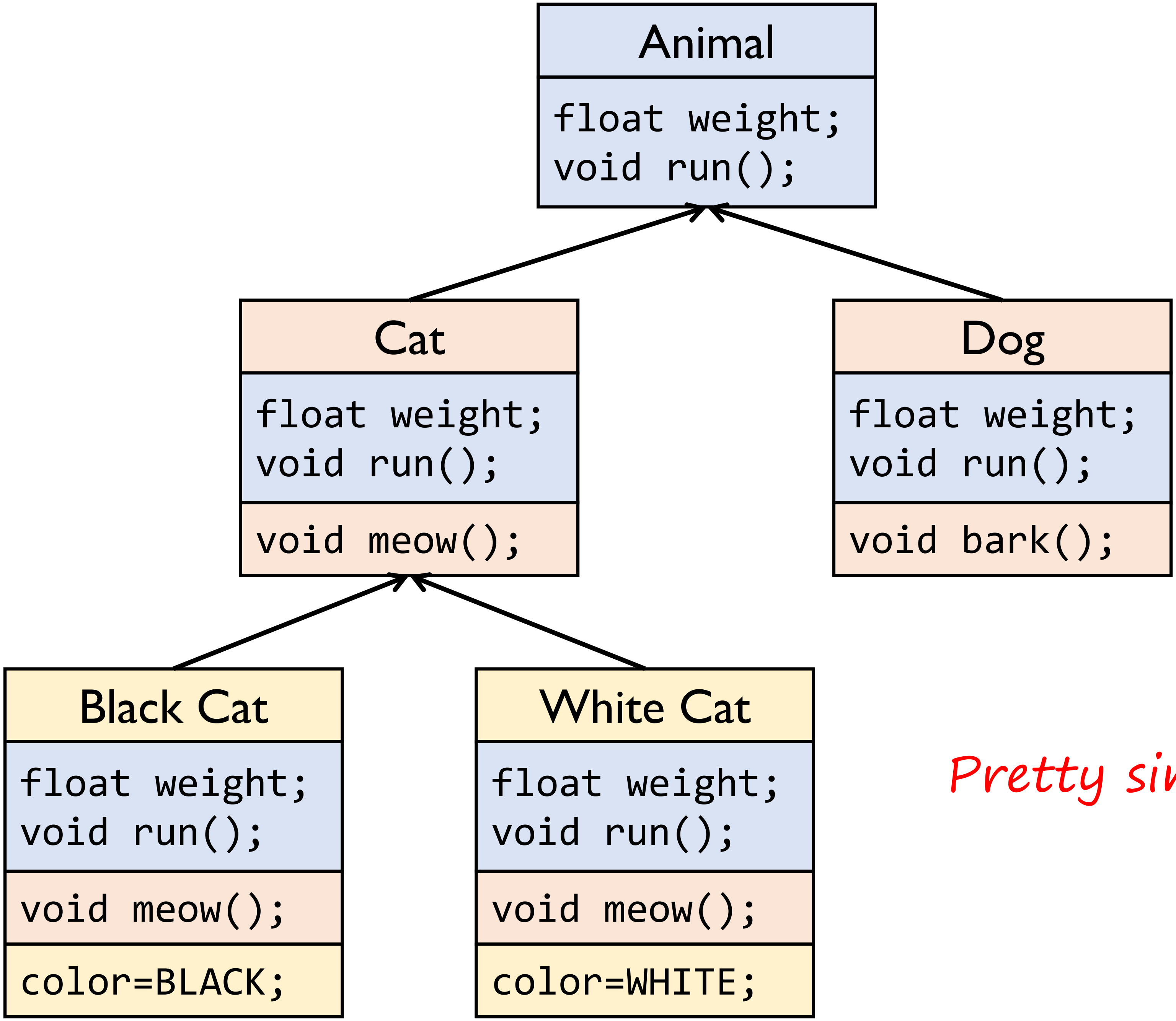
Prototypal Inheritance

Songchao Wang
Zibo Wang
Mingxiao An

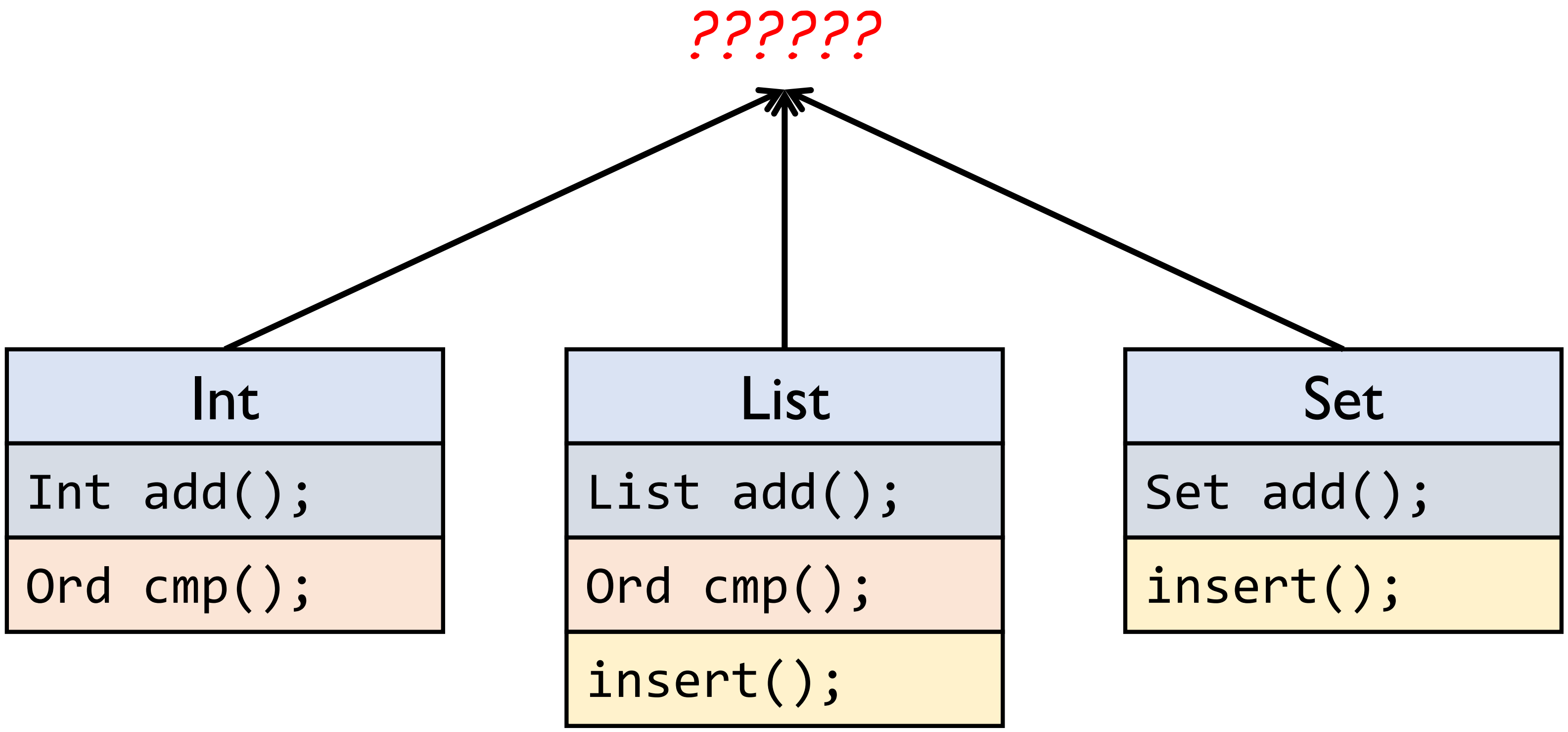
What is it?

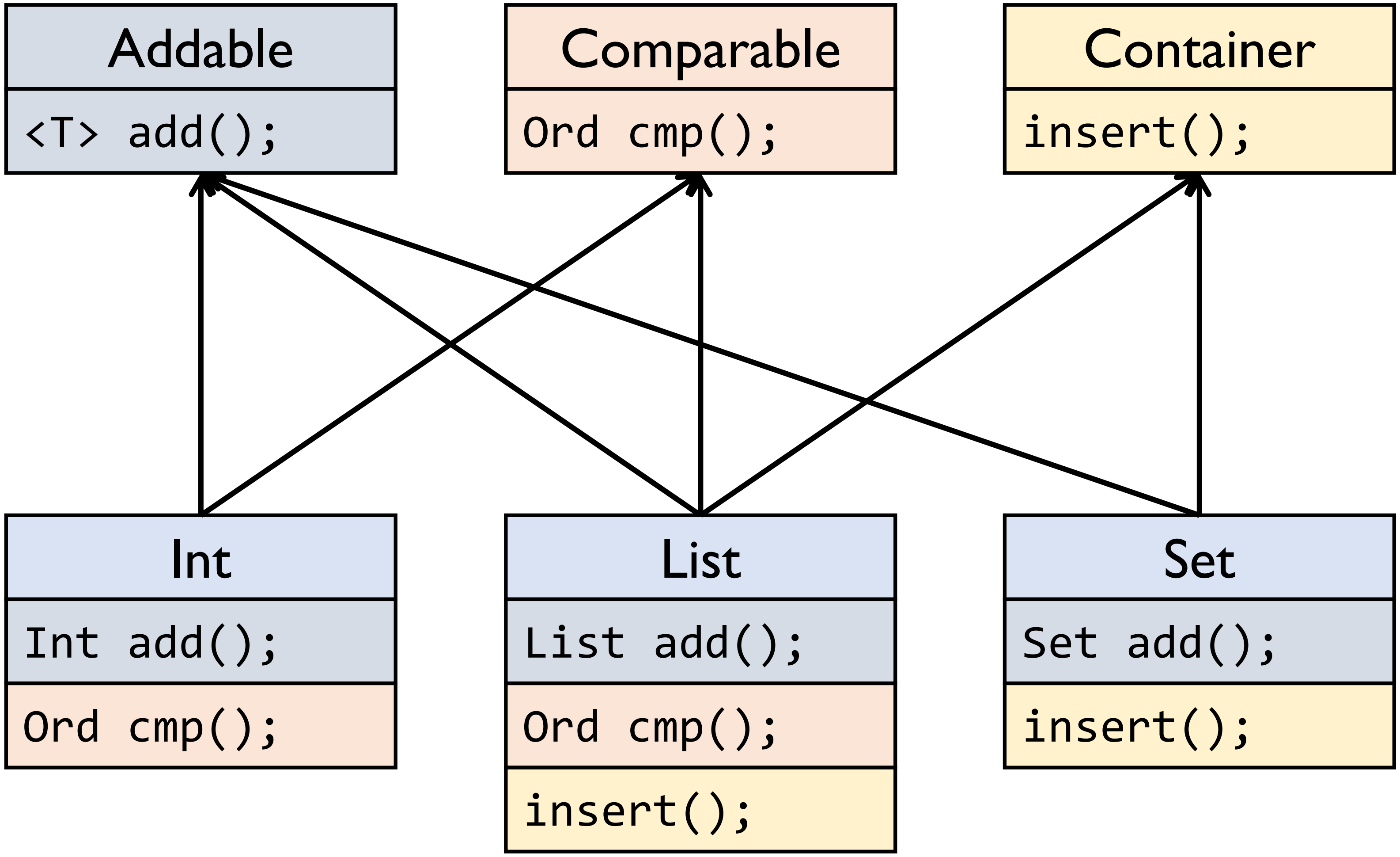
Prototypal Inheritance

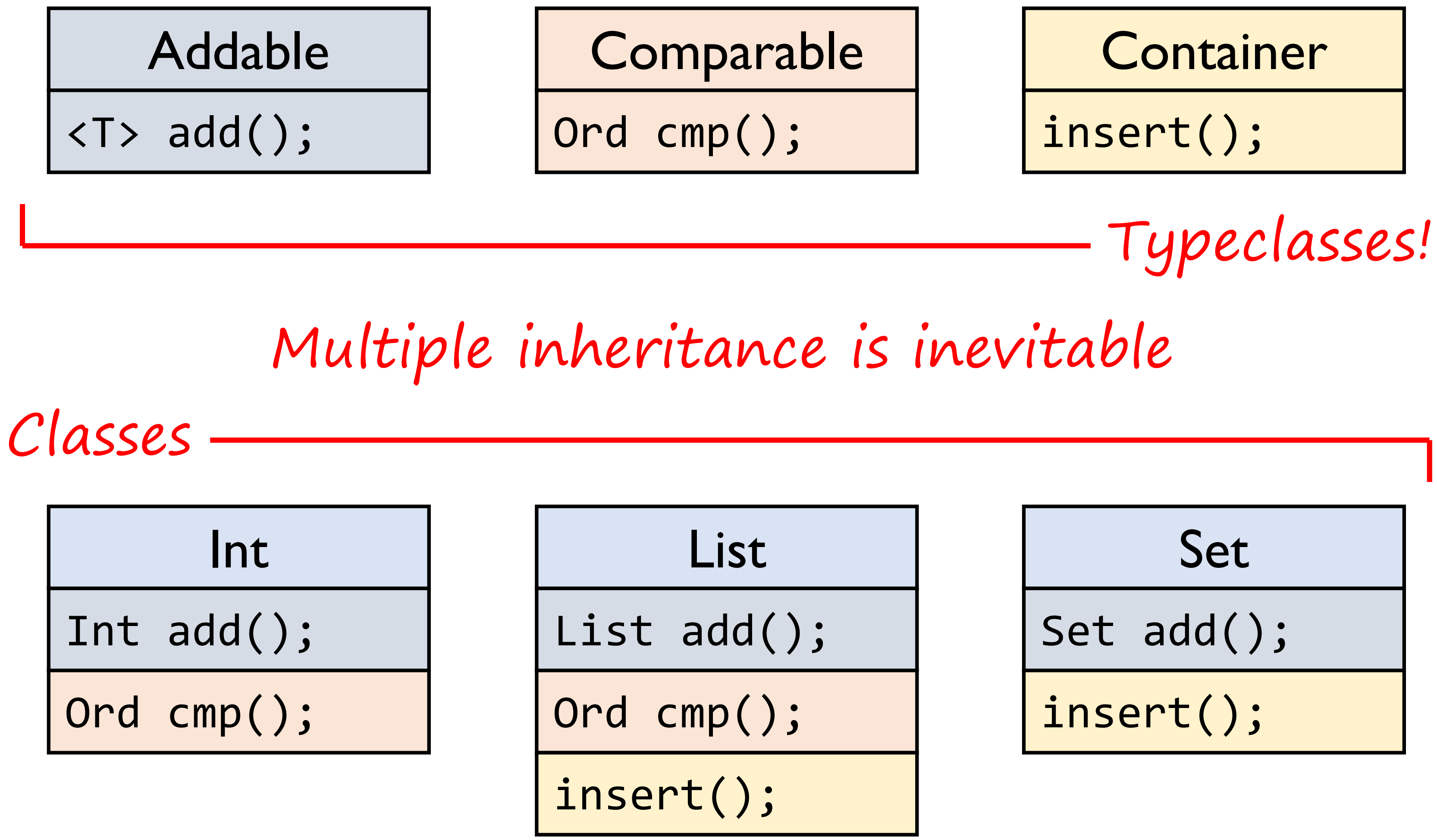
Songchao Wang
Zibo Wang
Mingxiao An

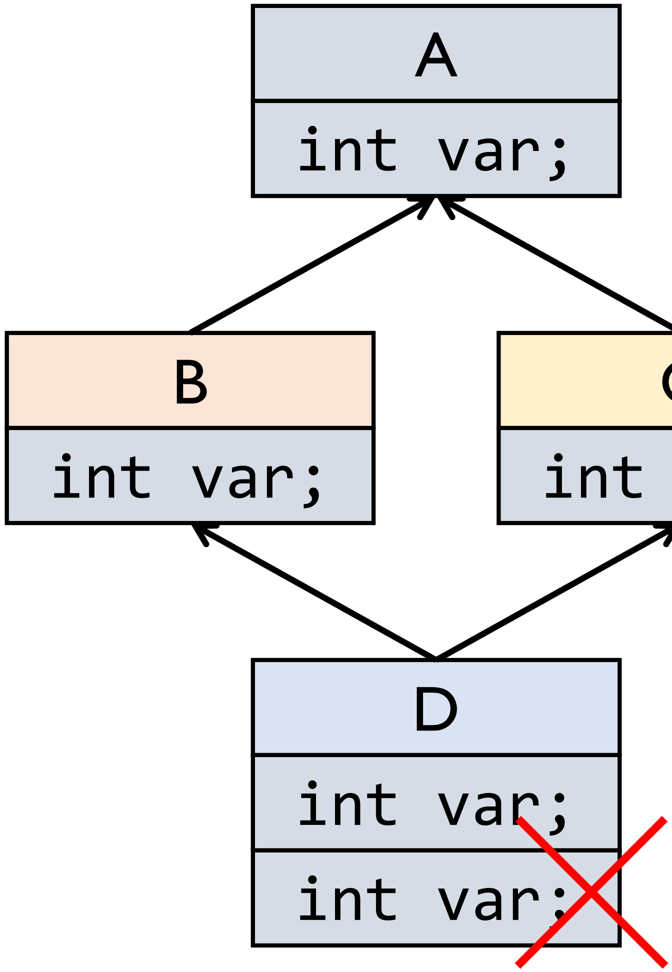


Pretty simple!

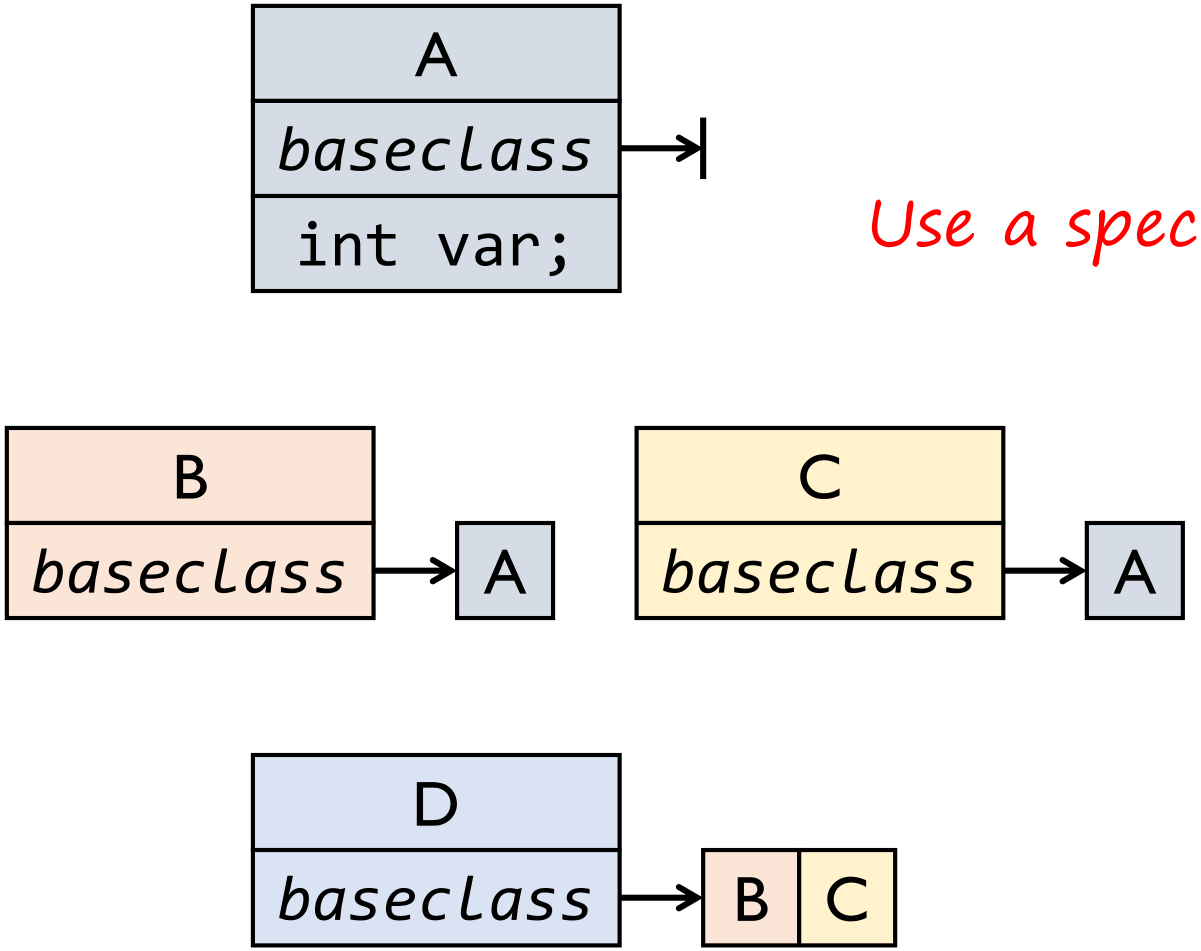

??????
Int
Int add();
Ord cmp();
List
List add();
Ord cmp();
insert();
Set
Set add();
insert();


Addable
<T> add();
Comparable
Ord cmp();
Container
insert();
Int
Int add();
Ord cmp();
List
List add();
Ord cmp();
insert();
Set
Set add();
insert();

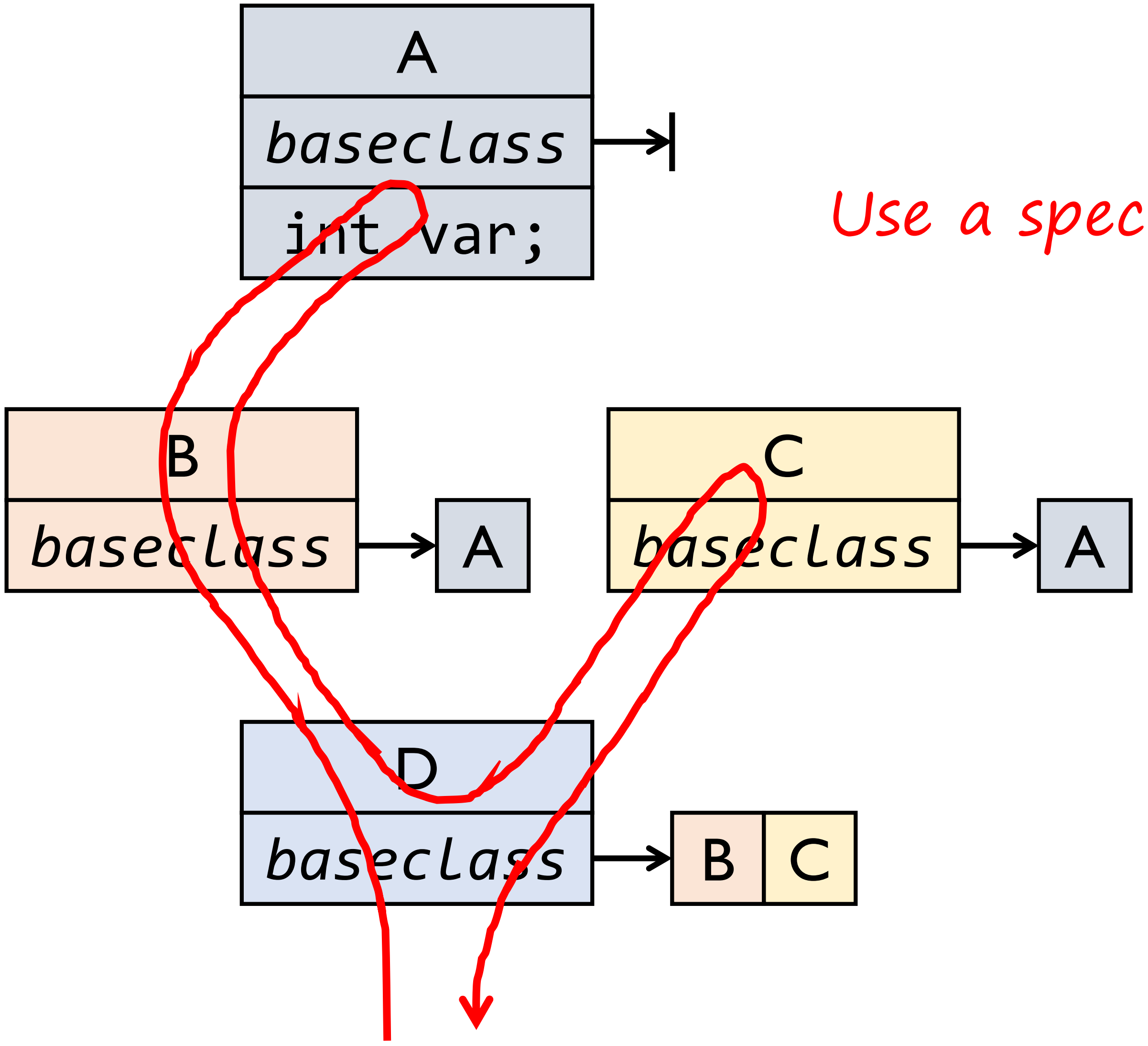

Addable
<T> add();
Comparable
Ord cmp();
Container
insert();
Typeclasses!
Multiple inheritance is inevitable
Classes
Int
Int add();
Ord cmp();
List
List add();
Ord cmp();
insert();
Set
Set add();
insert();



How to solve the diamond inheritance problem?

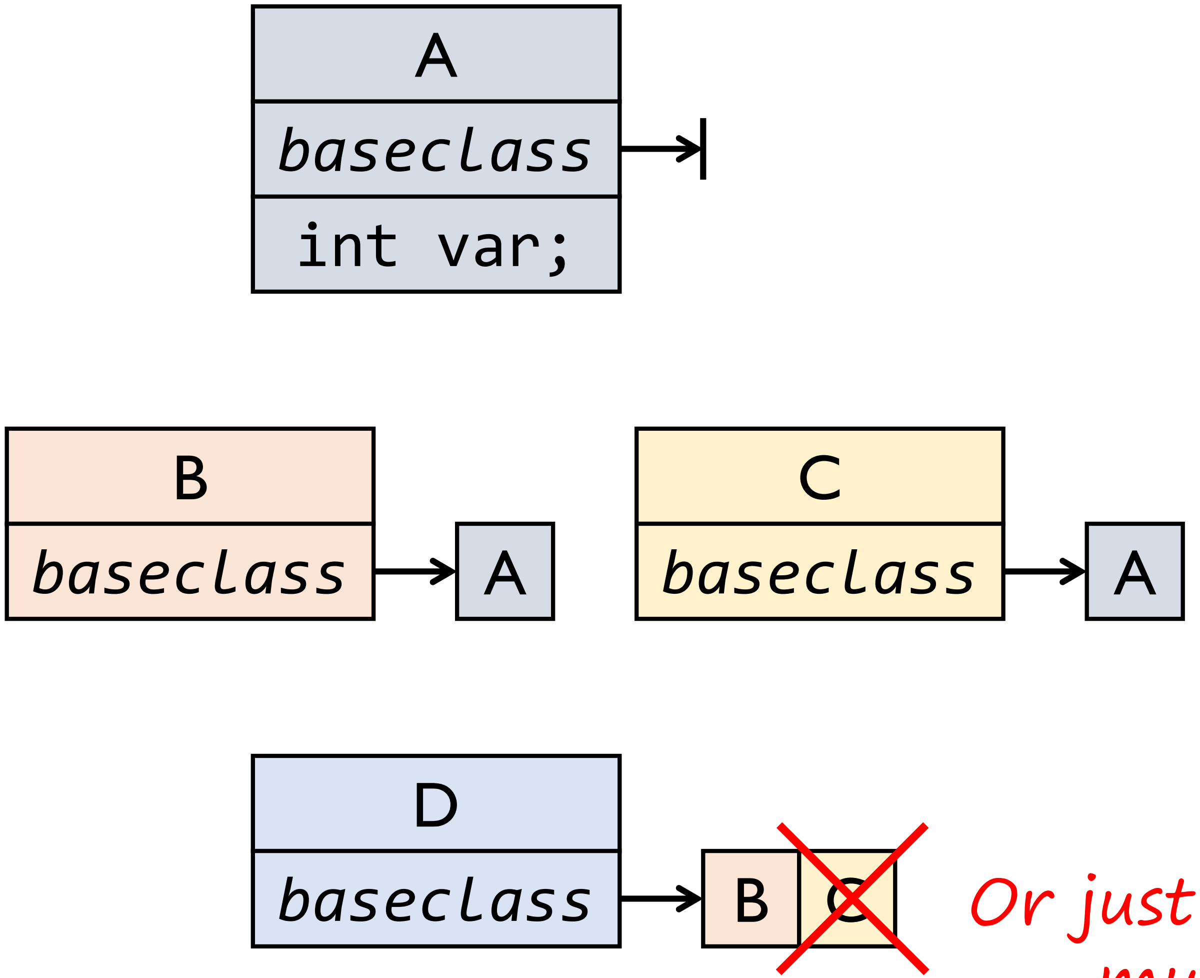


Use a special reference...

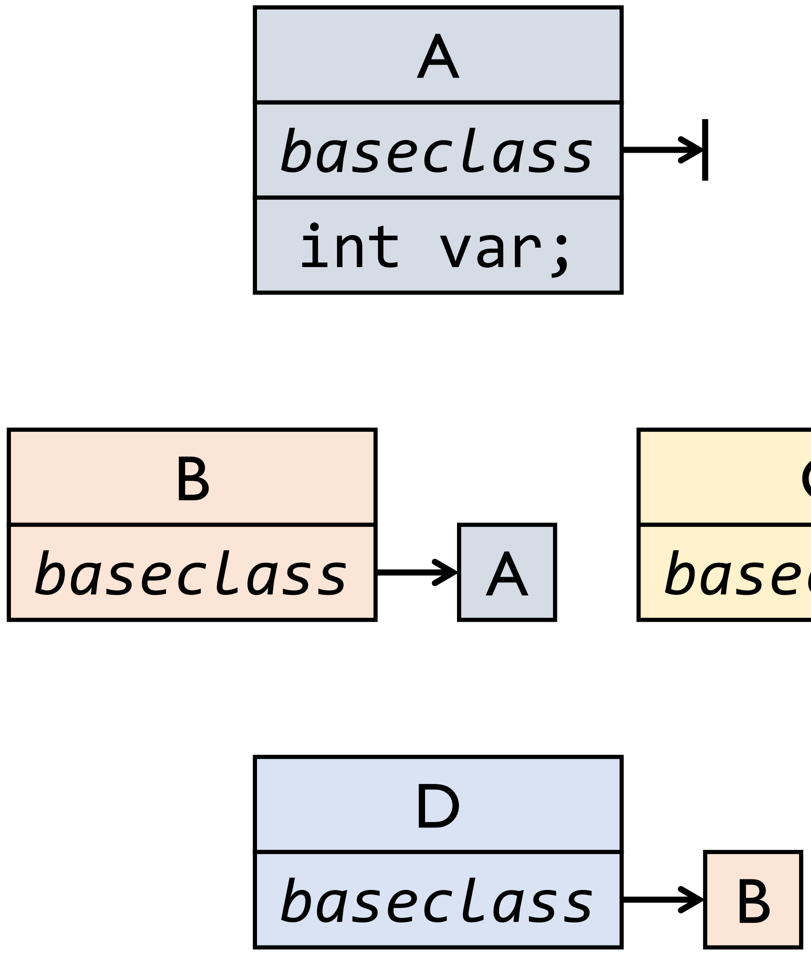


Use a special reference...

... and search the attributes using DFS!



Or just forbid multiple inheritance?



Then we need to provide
a way to join two classes
(at the runtime)

```
A = class(Object, …);
B = class(A, …);
C = class(A, …);

D = class(B, …);
join(D, C);
```

So the classes are dynamic now!

Dynamic classes are just normal objects

```
A = class(Object, …);
B = class(A, …);
C = class(A, …);

D = class(B, …);
join(D, C);
```

Create an instance of D $\rightarrow$

```
A = copy(Class);
modify A as we need
B = copy(A);
modify B as we need
C = copy(A);
modify C as we need

D = copy(B);
join(D, C);
modify D as we need

d = D();
```

Dynamic classes are just normal objects

```
A = class(Object, …);
B = class(A, …);
C = class(A, …);

D = class(B, …);
join(D, C);
```

Why not still use copy?

```
A = copy(Class);
modify A as we need
B = copy(A);
modify B as we need
C = copy(A);
modify C as we need

D = copy(B);
join(D, C);
modify D as we need

d = copy(D);
```

Prototypal Inheritance!

Everything is an object

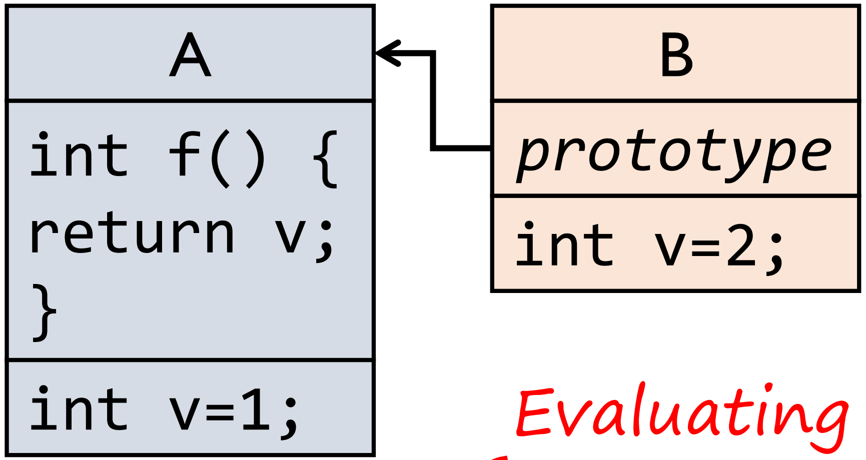
Use copy to inherit

… and instantiate

```
A = copy(Class);
modify A as we need
B = copy(A);
modify B as we need
C = copy(A);
modify C as we need

D = copy(B);
join(D, C);
modify D as we need

d = copy(D);
```



```
int f() {
return v;
}
int v=1;
```

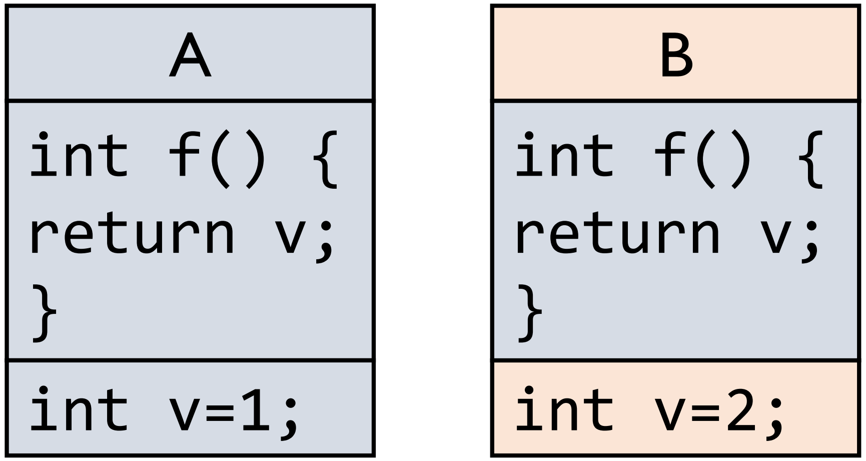
```
int v=2;
```

```
B.f() -> 2
A.f = {return v+1;}
B.f() -> 3
```

Evaluating A.f in the context of B

Delegation and Concatenation

copy made a real copy



```
int f() {
return v;
}
int v=1;
```

```
int f() {
return v;
}
int v=2;
```

```
B.f() -> 2
A.f = {return v+1;}
B.f() -> 2
```

Concatenation

```
alice = people copy name: 'Alice'.
```

Delegation

```
bob = (| parent* = people. name = 'Bob'. |).
```

```
var foo = {a: 1, b: 2};
var bar = {a: 0, c: 3};
Object.setPrototypeOf(bar, foo);
bar.a; -> 0
bar.b; -> 2
bar.c; -> 3
foo.b = 4;
bar.b; -> 4
bar.b = 5;
foo.b; -> 4
```

can be replaced by

```
var bar = Object.create(foo);
bar.a = 0;
bar.c = 3;
```

Now foo looks more like a class!

```
template = {
    value = 0,
    func = function() return 10 end
}
a = {my_value = 1}
setmetatable(a, template)
a.func() -> 10
a.value -> 0
a.my_value -> 1
```

```
template = {
    value = 0,
    func = functio
}
a = {my_value = 1}
setmetatable(a, te
a.func() -> 10
a.value -> 0
a.my_value -> 1
```

```
function Account:new (o)
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    return o
end
a = Account:new{balance = 0}
a:deposit(100)
```

Now Account looks more like a class!

How to do these cool things in python™

How to do these cool things in python™

```
class object:

    def __init__(self):
        self.__parent__ = type(self)

    def __call__(self):
        obj = type(self)()
        obj.__parent__ = self
        return obj

    def __getattr__(self, name):
        return getattr(self.__parent__, name)
```

a=object() will set a.__parent__ to object

b=a() will set a as b's parent

when attributes not found, look for parents' namespace

How to do these cool things in python™

```
class object:

    def __init__(self):
        self.__parent__ = type(self)

    def __call__(self):
        obj = type(self)()
        obj.__parent__ = self
        return obj

    def __getattr__(self, name):
        return getattr(self.__parent__, name)
```

```
a = object()
a.foo = 1
a.bar = 2
b = a()
b.foo -> 1
b.bar = 3
b.bar -> 3
a.bar -> 2
a.foo = 4
b.foo -> 4
c = b()
c.bar -> 3
```

Learn the idea, not the language!

So how to determine whether a language is prototype-based?

Dynamic type system

"type" is just normal objects

Everything is an object

usually a dictionary

New object comes from copying old object

usually with a special "prototype" attribute

Flexible.

Easy to design.

Dynamic type system

Unsafe! Hard to optimize!
Lots of attribute-looking-up overhead!

Everything is an object

Beautiful. Neat.

Unfamiliar! Hard to program!

New object comes from copying old object

Easy to adjust the structure.

Thanks!

With the help of:
Wikipedia,
Stack overflow,
Self website,
Lua website,
ECMAScript website,
and of course Python documents.

Slides' overall style comes from:
@ScottWlaschin's *Functional Design Patterns*.