Course 142A Compilers & Interpreters
Object-Oriented Runtime Support

Lecture Week 7

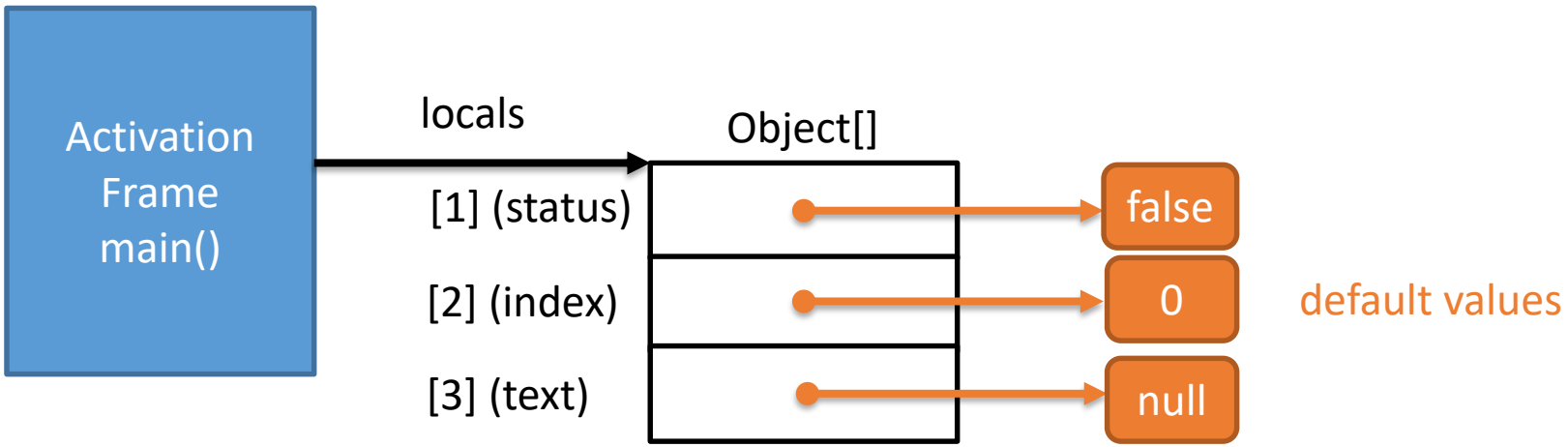
Prof. Dr. Luc Bläser

Last Lecture - Quiz

```
void main() {
  boolean status;
  int index;
  string text;
}
```

How do we manage these variables in the interpreter?

Activation Frame



Why do we default-initialize variables?

Security Measures

- Always initialize variables (even locals)
- Boundary checks (null deref, array index)
- Stack overflow/underflow detection
- Check correct bytecode and type consistency
- Compatibility with external assemblies (not here)
- Garbage Collection (later)

Checks in Interpreter

iadd

```
int right = checkInt(pop());
int left = checkInt(pop());
push(left + right);
```

Must be int type

No stack underflow

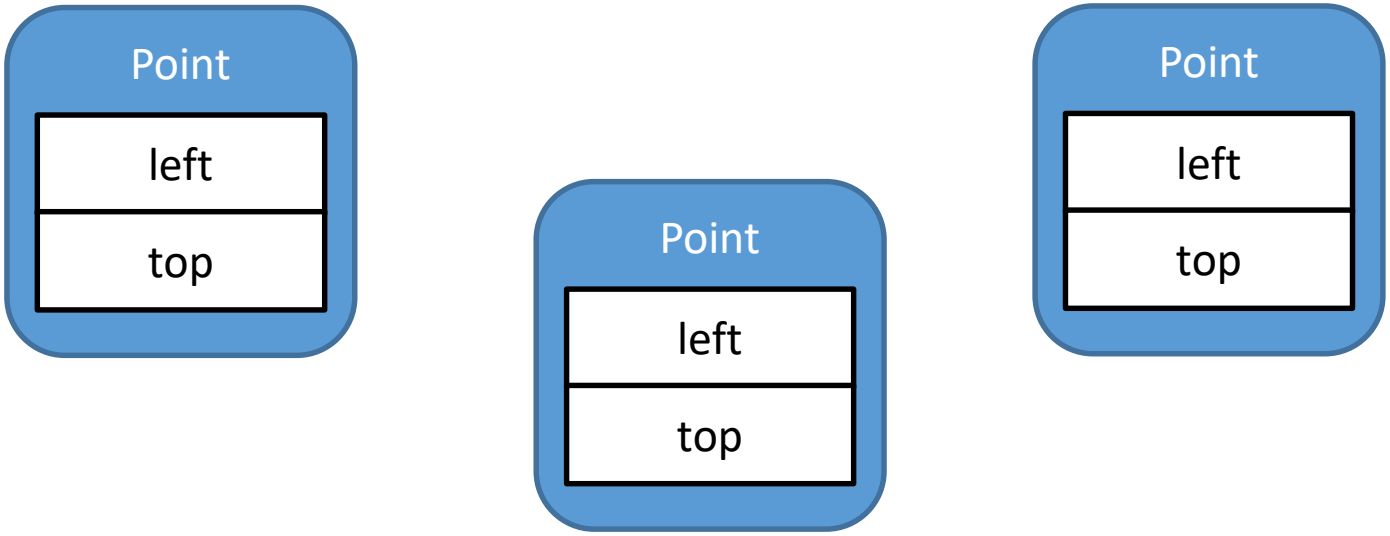
Next Step

```
class Point {
  int left;
  int top;
}
```

How should the interpreter manage these variables?

Object Support

- Heap with objects containing fields



Do we feature static fields or static methods in UCI-Java?

Object-Orientation

What do we need to support for object-orientation?

OO-Support

Impacts all system components

Compiler:

- Lexer and Parser
- Semantic Checker
- Code Generator

Virtual Machine:

- Loader
- Interpreter
- Heap & GC

Today's Topics

- Object-Orientation
- Heap Management
- Objects, Arrays, and References
- Single Inheritance

Learning Goals

- Know the compiler and VM features to support object-orientation
- Understand principals of object management in the runtime system

OO in Lexer & Parser

- new-operator for classes (and arrays)
  - `new Point();`
- Indirect access expressions (designators)
  - `x.y.z`
  - `x.y.F()`
- Type cast and instanceof-operator
  - `(Node)x`
  - `x instanceof Node`
- Base class
  - `class Sub extends Base {}`

Done!

OO in Semantic Checker (1)

- Assignment-compatibility of types
  - null is compatible to all reference types
  - Subclass is compatible to base class (implicit up-cast)
- Assignments/parameter passing
  - Types are assignment-compatible
- Comparisons a == b and a != b
  - a is assignment-compatible to b, or vice versa (b to a)

Done!

OO in Semantic Checker (2)

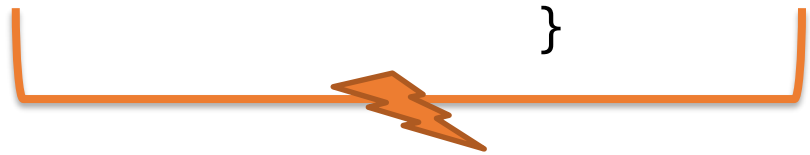
- No cyclic inheritance

```
class A extends B { … }

class B extends A { … }
```

- Overriding matches
  - Same signature and return type

```
class Base {
  void f(int x) { … }
}
```

```
class Sub : Base {
  int f(string s) { … }
}
```



Done!

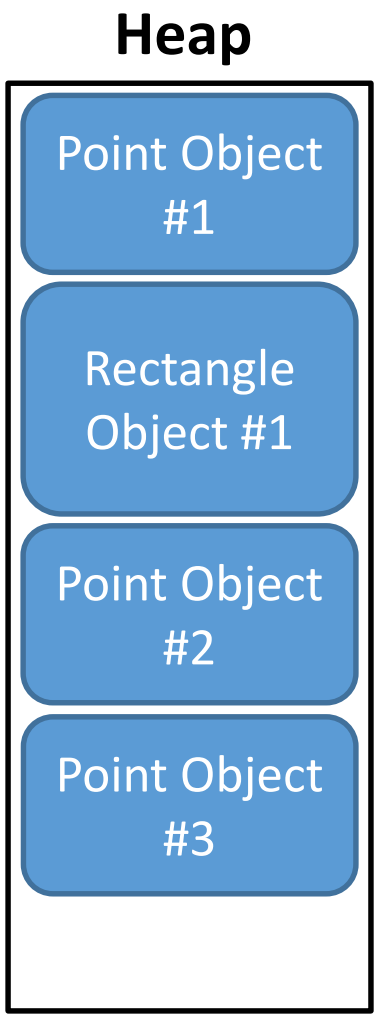
Language Features: Out of Scope

- Constructor and initializer
  - Not in UCI-Java
- Access modifier (private, public)
  - Not in UCI-Java
- Inheritance and type polymorphism
  - Later
- Garbage collection
  - Later

Objects at Runtime

- Store created objects on heap
- Heap = linear address space

```
a = new Point();
b = new Rectangle();
c = new Point();
d = new Point();
```



Stack vs. Heap

Why are objects not stored on the stack?

Object Lifetime

- No bound to method incarnation

```
Point getPoint() {
  return new Point();
}
```

Method Escape

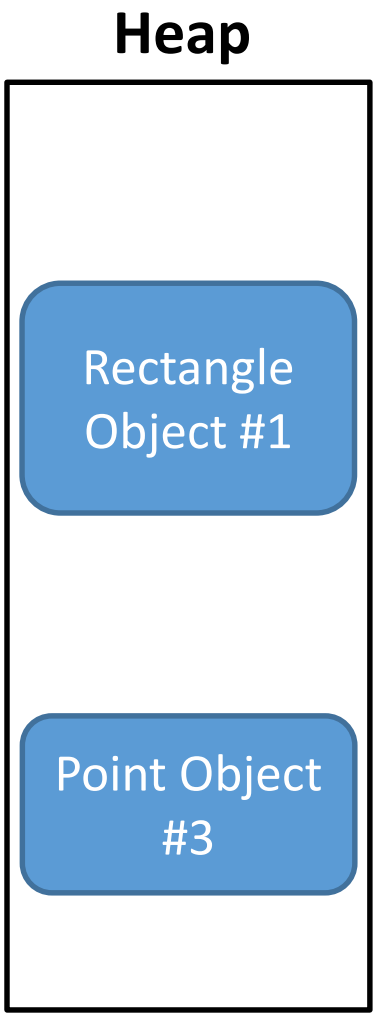
```
void setup(Rectangle r) {
  r.topLeft = new Point();
}
```

Deallocation

- No hierarchy among objects
- Disposal can be in any order
- Deallocation leads to holes

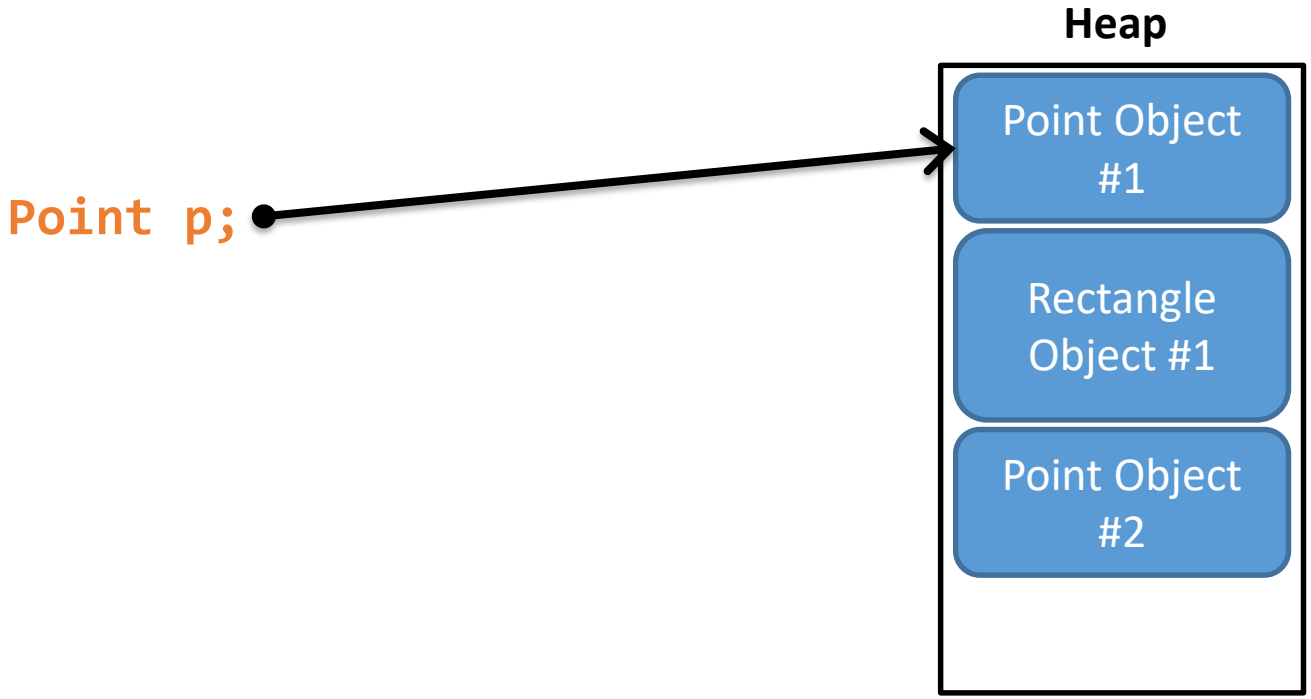
```
delete a
delete c
```

Garbage Collection covered later



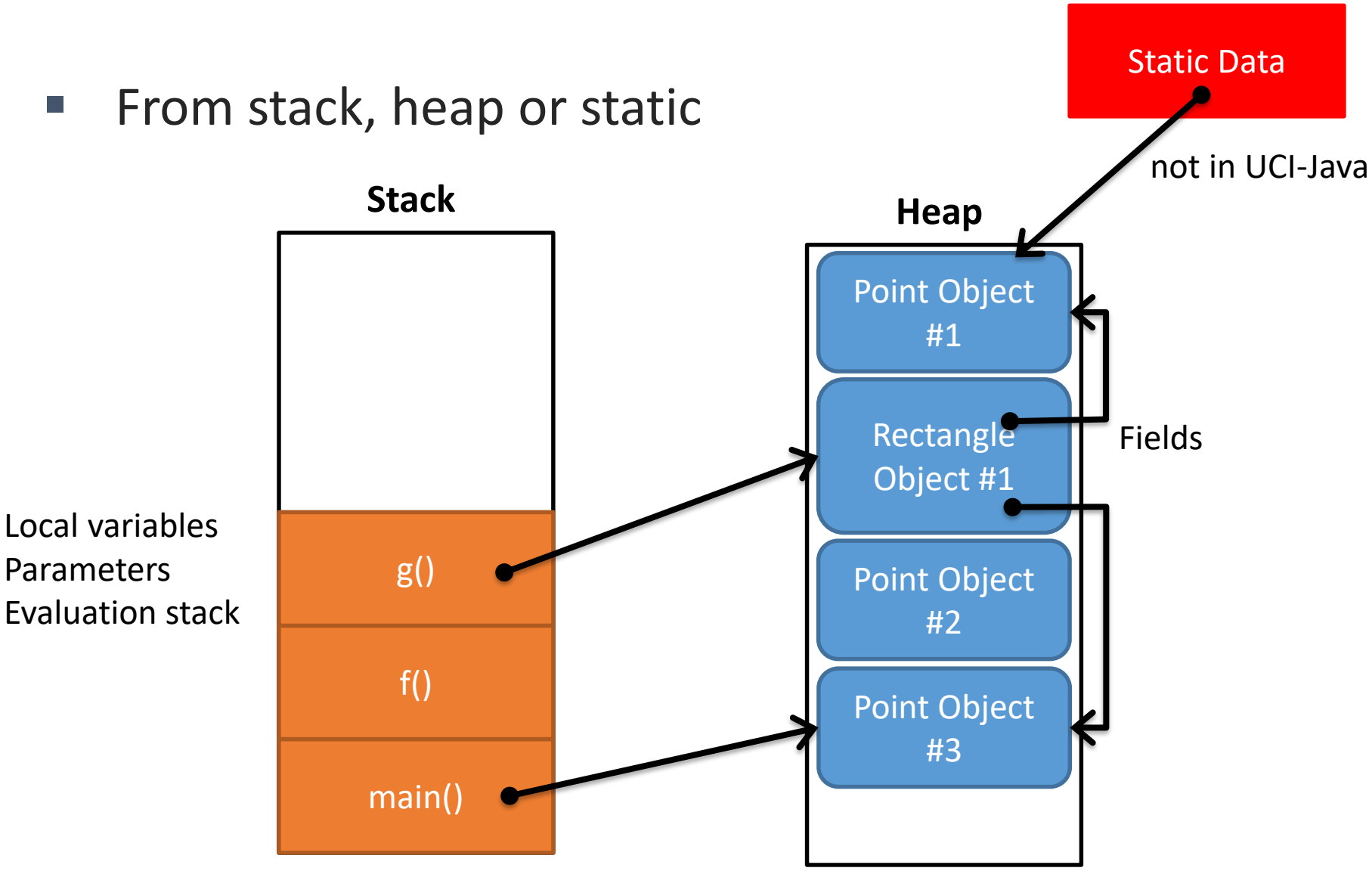
Object References

- Pointer to an object in heap
  - Represented as object's memory address
- Variables only store the reference
  - If reference type, i.e. class, array, string



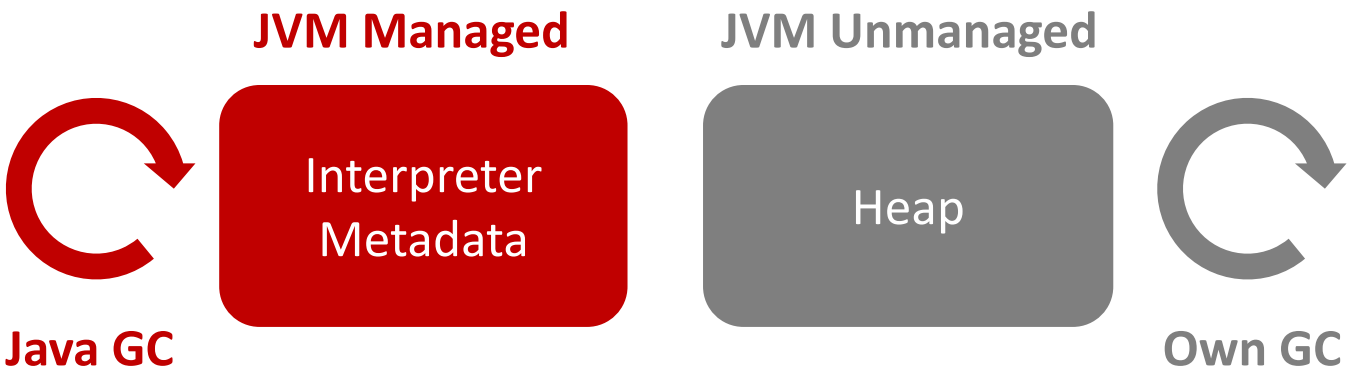
Sources of References

- From stack, heap or static



Heap Design

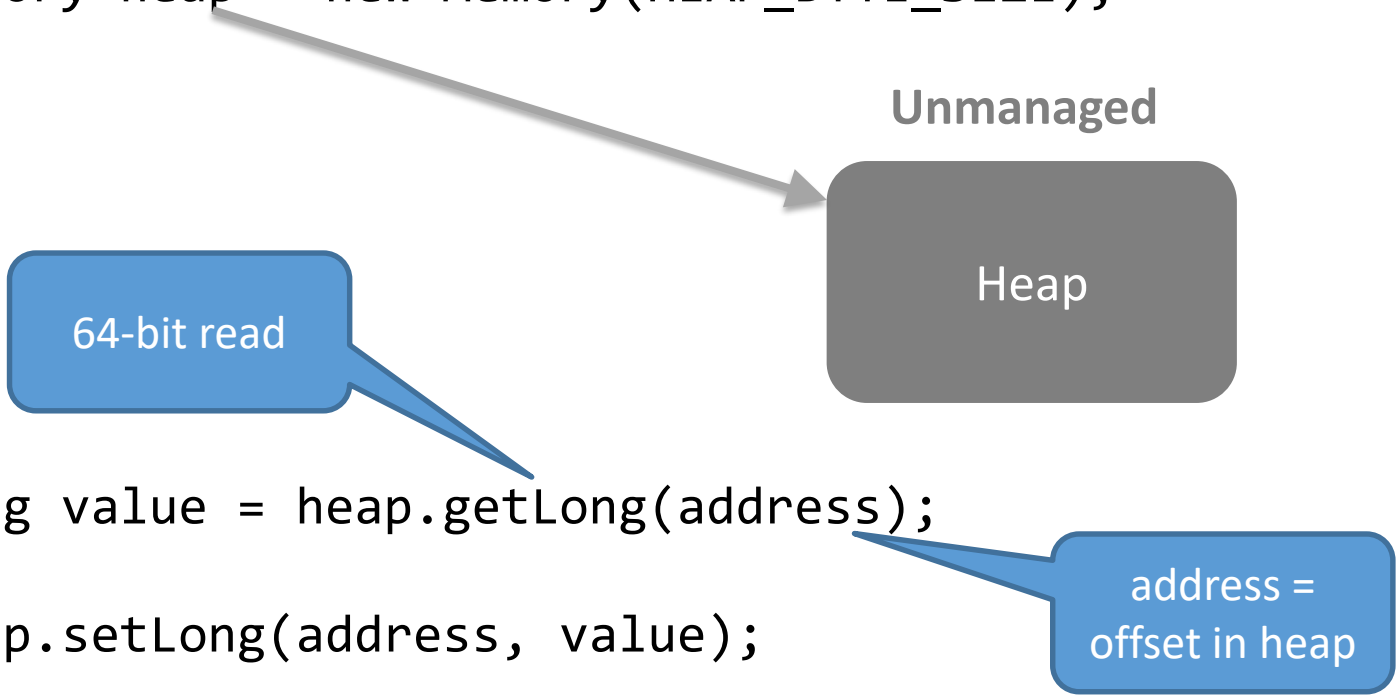
- Custom raw heap in unmanaged memory
- Java Native Access



Unmanaged Memory in Java

- Allocate memory in Java Native Access

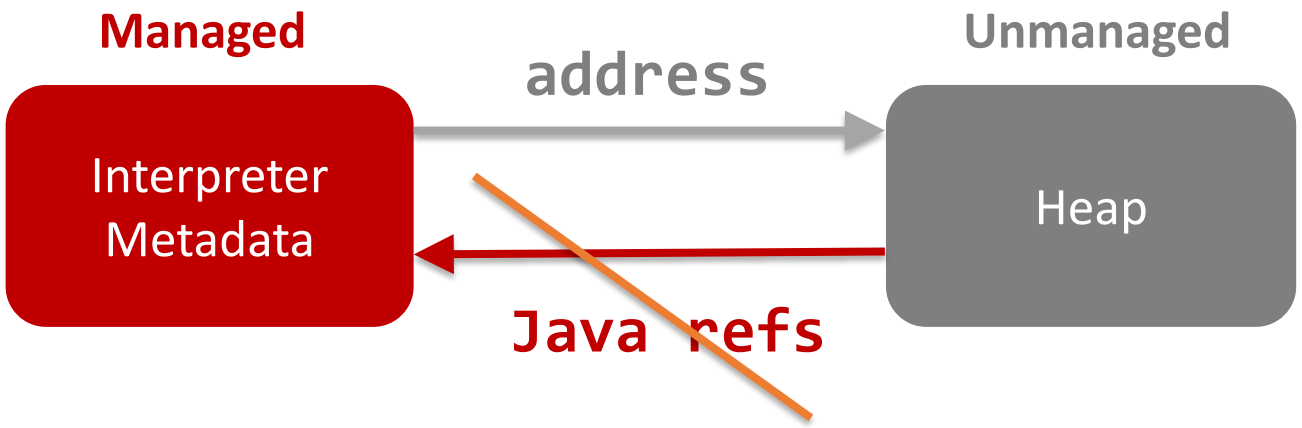
```
import com.sun.jna.Memory;

Memory heap = new Memory(HEAP_BYTE_SIZE);
```




```
long value = heap.getLong(address);
…
heap.setLong(address, value);
```

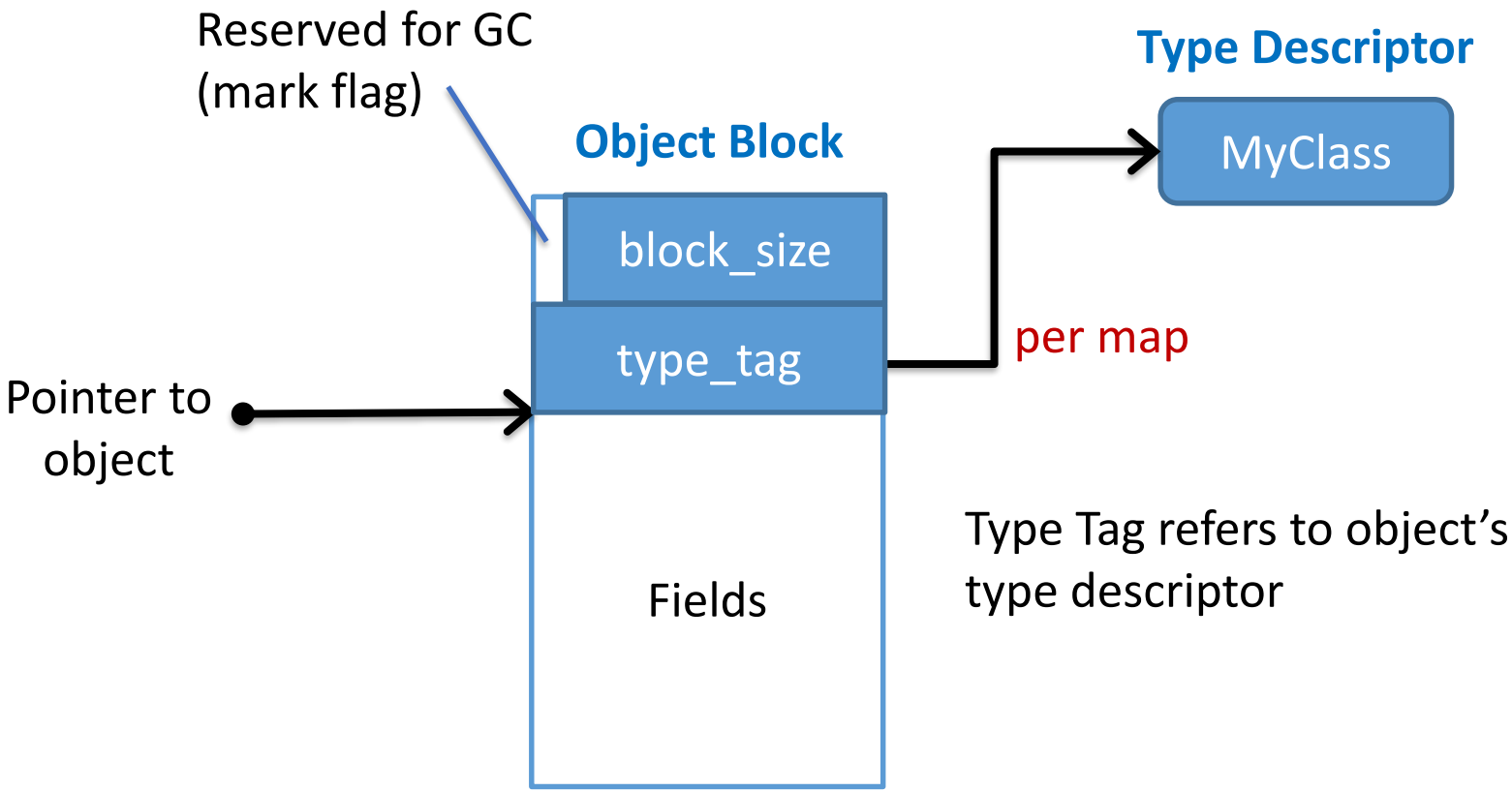
Raw Heap: Principles

- 64-bit addresses/pointers (8 bytes)
- Raw heap cannot store Java references



Use map instead (e.g. HashMap<Long, Object>) when needed

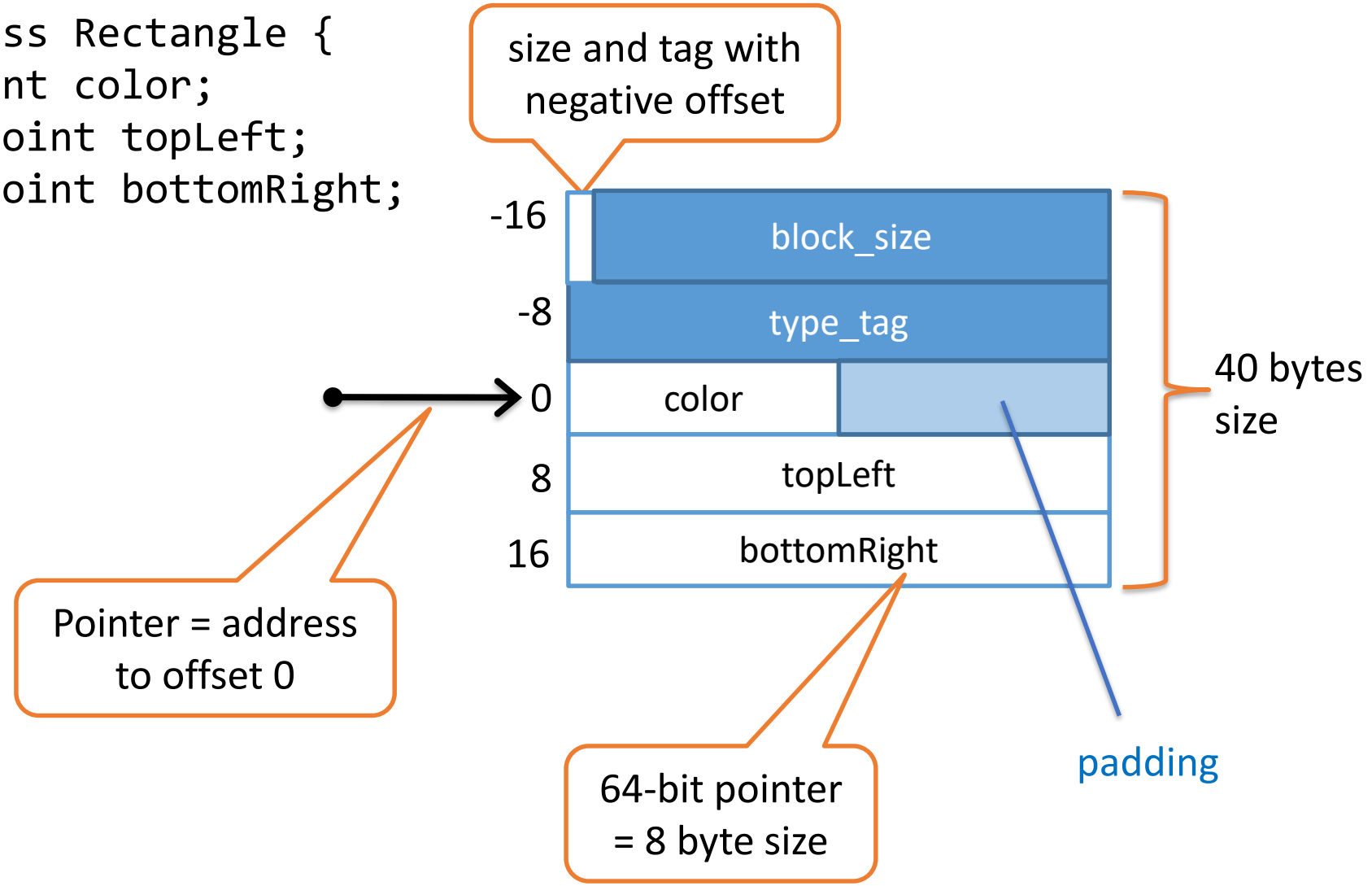
Object Layout



Simplicity: Keep descriptors in managed JVM heap

Concrete Object Layout

```
class Rectangle {
  int color;
  Point topLeft;
  Point bottomRight;
}
```



Heap

- Convenience class for use by the interpreter

```
class Heap {
  …
  Pointer allocateObject(ClassDescriptor type);

  Object readField(Pointer ptr, int index);

  void writeField(Pointer ptr, int index, Object value);
  …
}
```

Wrapper for long address

Fields are numbered

boolean, int, String, Pointer

OO Interpretation Patterns

new

```
var type = (ClassDescriptor)instruction.getOperand();
var instance = heap.allocateObject(type);
push(instance);
```

Done!

aconst_null

```
push(null);
```

Open!

OO Interpretationsmuster

getfield

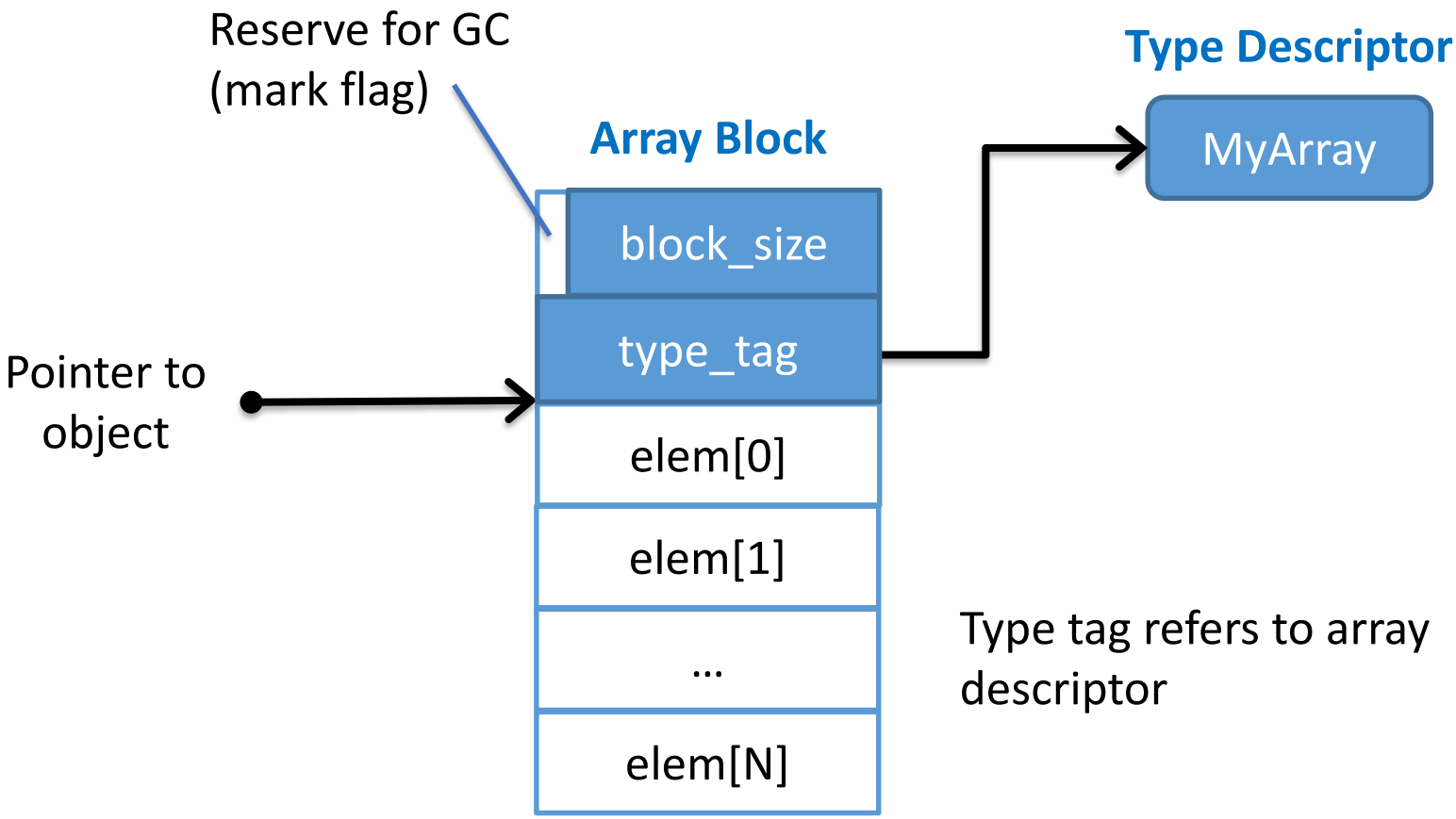
```
var field = (FieldDescriptor)instruction.getOperand();
var index = field.getIndex();
var instance = pop();
var value = heap.readField(instance, index);
push(value);
```

Done!

putfield

```
var field = (FieldDescriptor)instruction.getOperand();
var index = field.getIndex();
var value = pop();
var instance = pop();
heap.writeField(instance, index, value);
```

Array Block



Simplicity: We can compute array length from block size (no padding)

Similar for Arrays

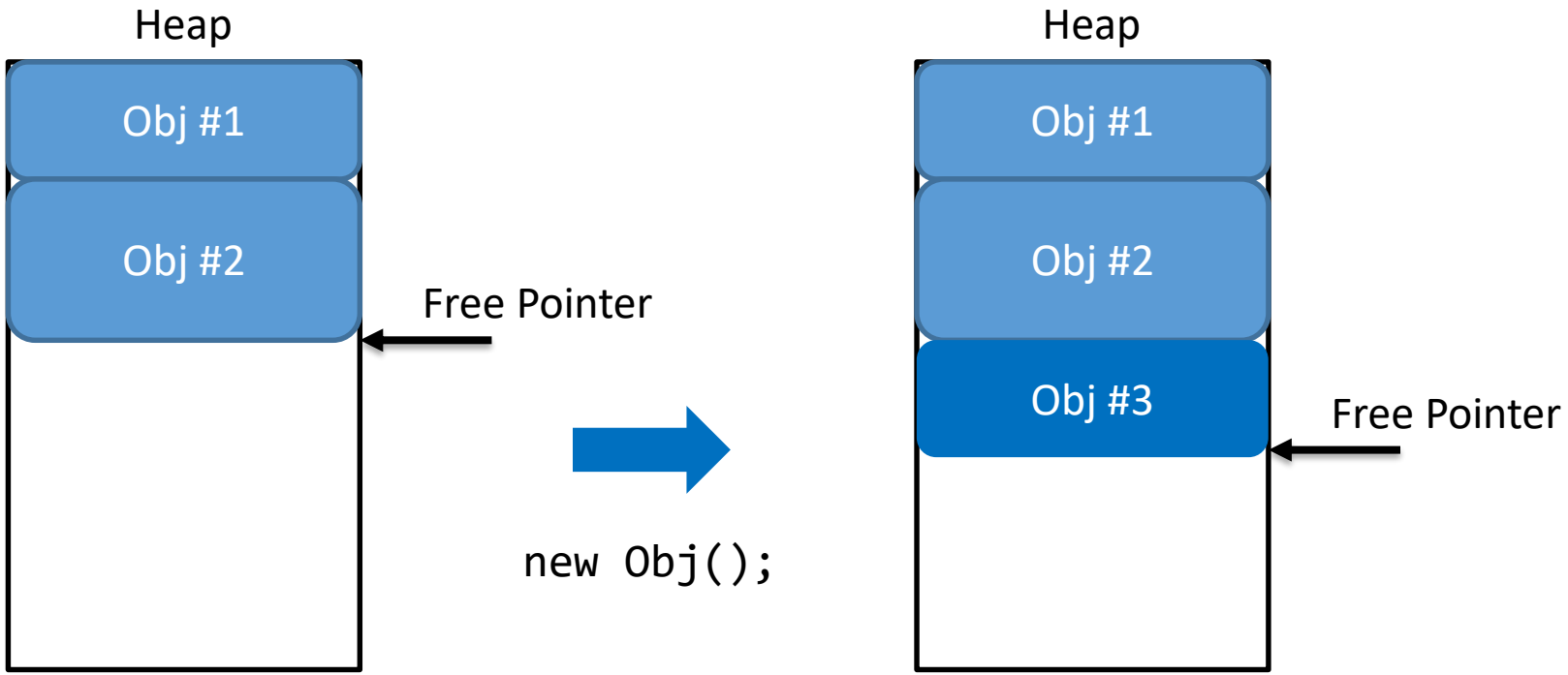
```
class Heap {
  …
  Pointer allocateArray(ArrayDescriptor type, int length);
  int getArrayLength(Pointer array);
  Object readElement(Pointer array, int index);
  void writeElement(Pointer array, int index, Object value);
}
```

newarray

```
var type = (ArrayDescriptor)instruction.getOperand();
var length = pop();
var array = heap.allocateArray(type, length);
push(array);
```

Simplest Heap

- Currently no deallocation yet (no GC)
- Simply allocate blocks linearly



Heap Allocation

```
Pointer allocateObject(ClassDescriptor type) {
  int size = 16 + type.getAllFields().length * 8;
  if (freePointer + size > limit) {
    throw new VMException("Out of Memory");
  }
  long address = freePointer;
  freePointer += size;
  heap.setLong(address, size);
  long typeTag = getDescriptorId(type);
  heap.setLong(address + 8, typeTag);
  address += 16;
  initialize(address, type);
  return new Pointer(address);
}
```

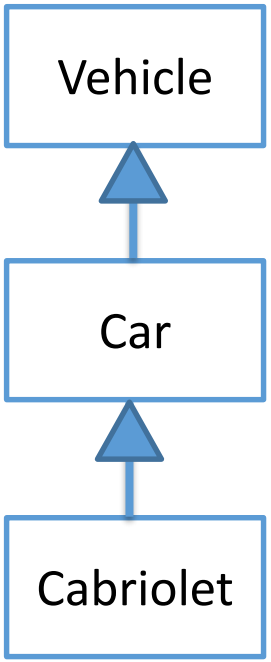
Simple layout

Header

Default initialization

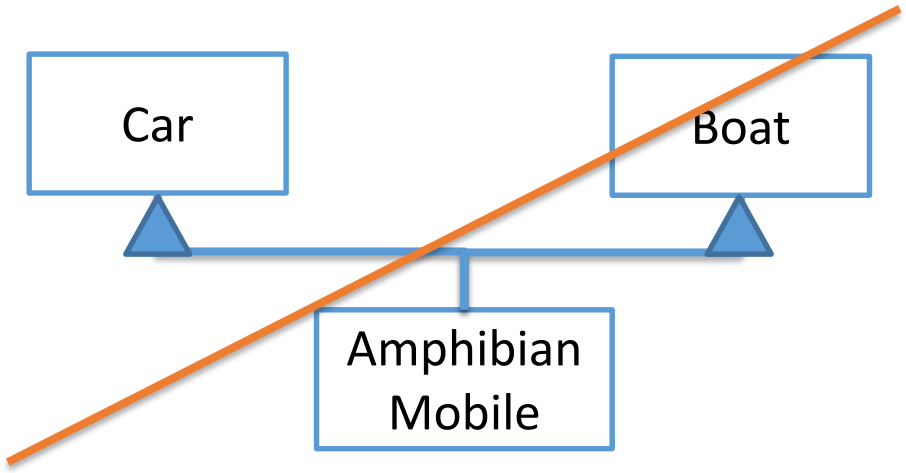
Inheritance

```
class Vehicle {
}

class Car extends Vehicle {
}

class Cabriolet extends Car {
}
```



Inheritance

- For simplicity, we only consider single inheritance (for the moment)



Which two aspects are combined with inheritance?

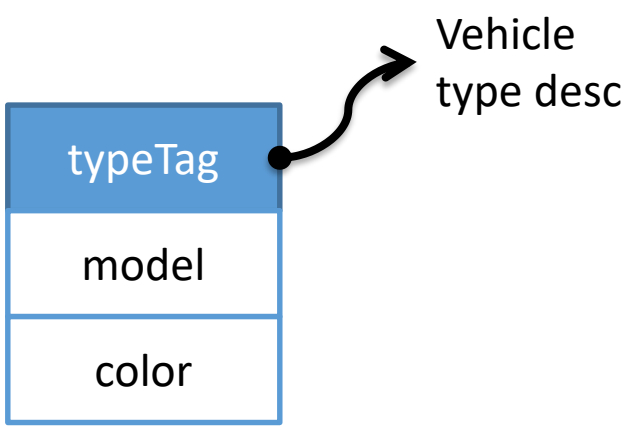
Inheritance: Two Aspects

- Code reuse
  - Subclass inherits fields & methods of base classes
- Type polymorphisms
  - Object of subclass also has the types of base classes

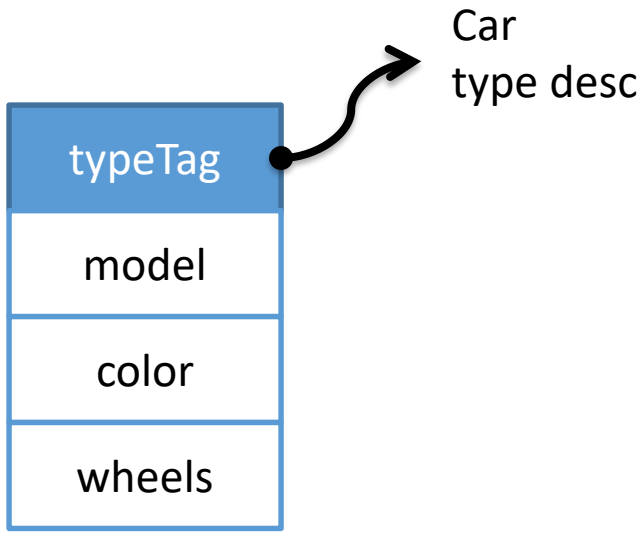
Code Reuse

- Subclass extends layout of the base class

```
class Vehicle {
  int model;
  int color;
}
```




```
class Car extends Vehicle {
  int wheels;
}
```

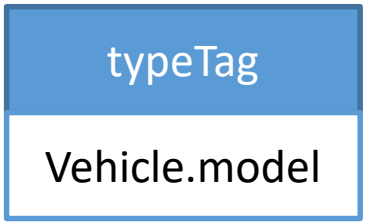


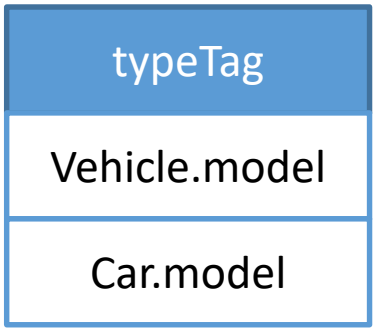
Variable Hiding

- Equally named variable in subclass = new definition

```
class Vehicle {
  int model;
}
```

```
class Car extends Vehicle {
  int model;
}
```





Static binding:
Code in Car uses Car.model
Code in Vehicle uses Vehicle.model

Field-Access

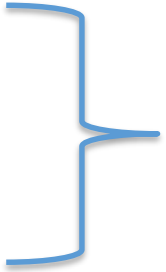
- Assign index to field descriptor

```
class Vehicle {
  string producer;    // index 0
  int model;          // index 1
}

class Car extends Vehicle {
  int wheels;         // index 2
}
```

```
getField <Car.wheels>   // index 2
putField <Car.model>    // index 1
```

Runtime Support for Inheritance

- Code Reuse
  - Inherit fields of base classes
- Type Polymorphism
  - Dynamic type casts/tests
  - Virtual method calls



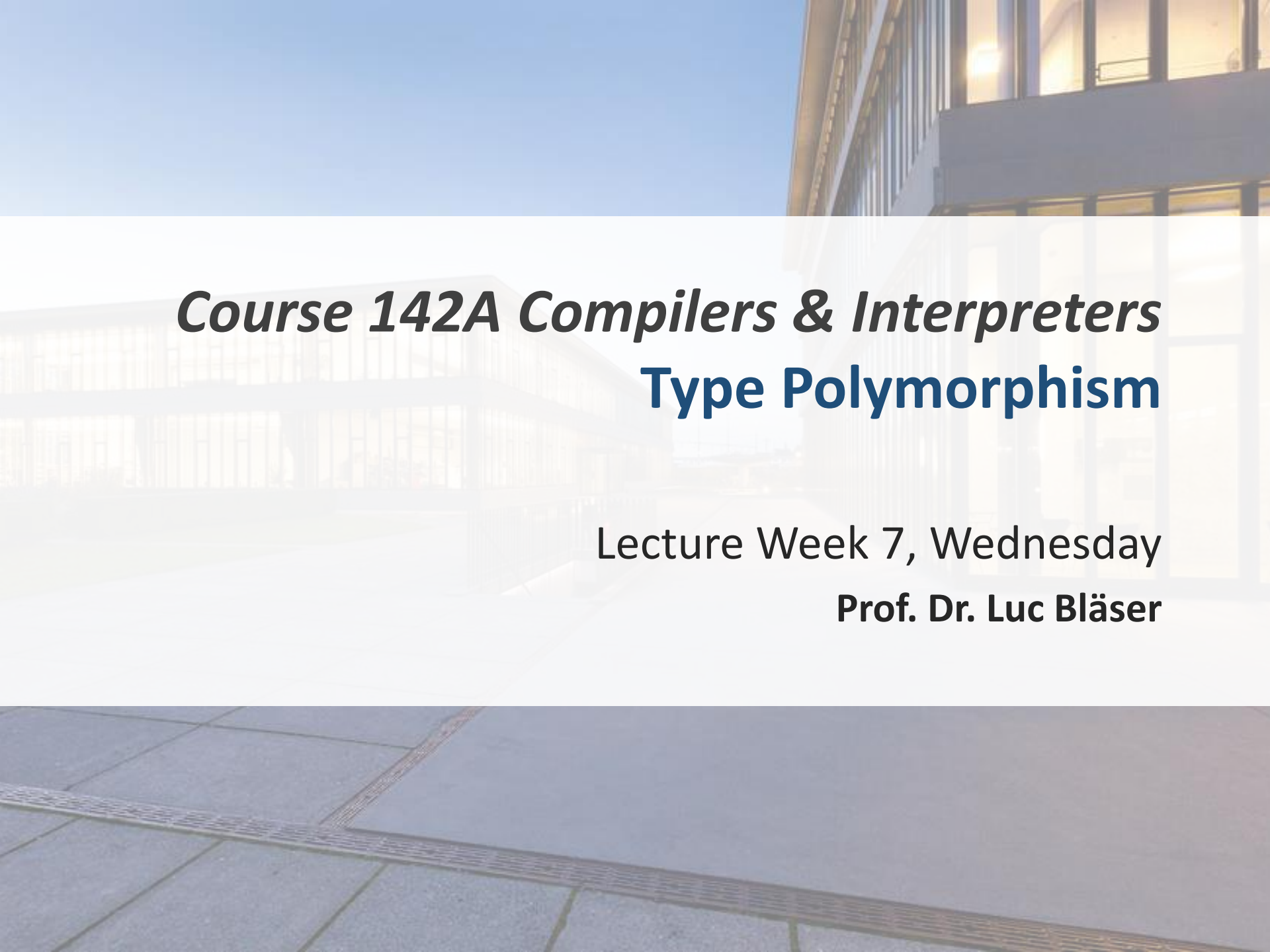
Next lecture

Review: Learning Goals

- ✓ Know the compiler and VM features to support object-orientation
- ✓ Understand principals of object management in the runtime system

Further Reading

- Dragon Book, Heap Management
  - Section 7.4
- Optional, if interested
  - B. Venners. Inside the Java Virtual Machine.
    - https://www.artima.com/insidejvm/ed2
    - Chapter 5 (JVM)



Course 142A Compilers & Interpreters
Type Polymorphism

Lecture Week 7, Wednesday

Prof. Dr. Luc Bläser

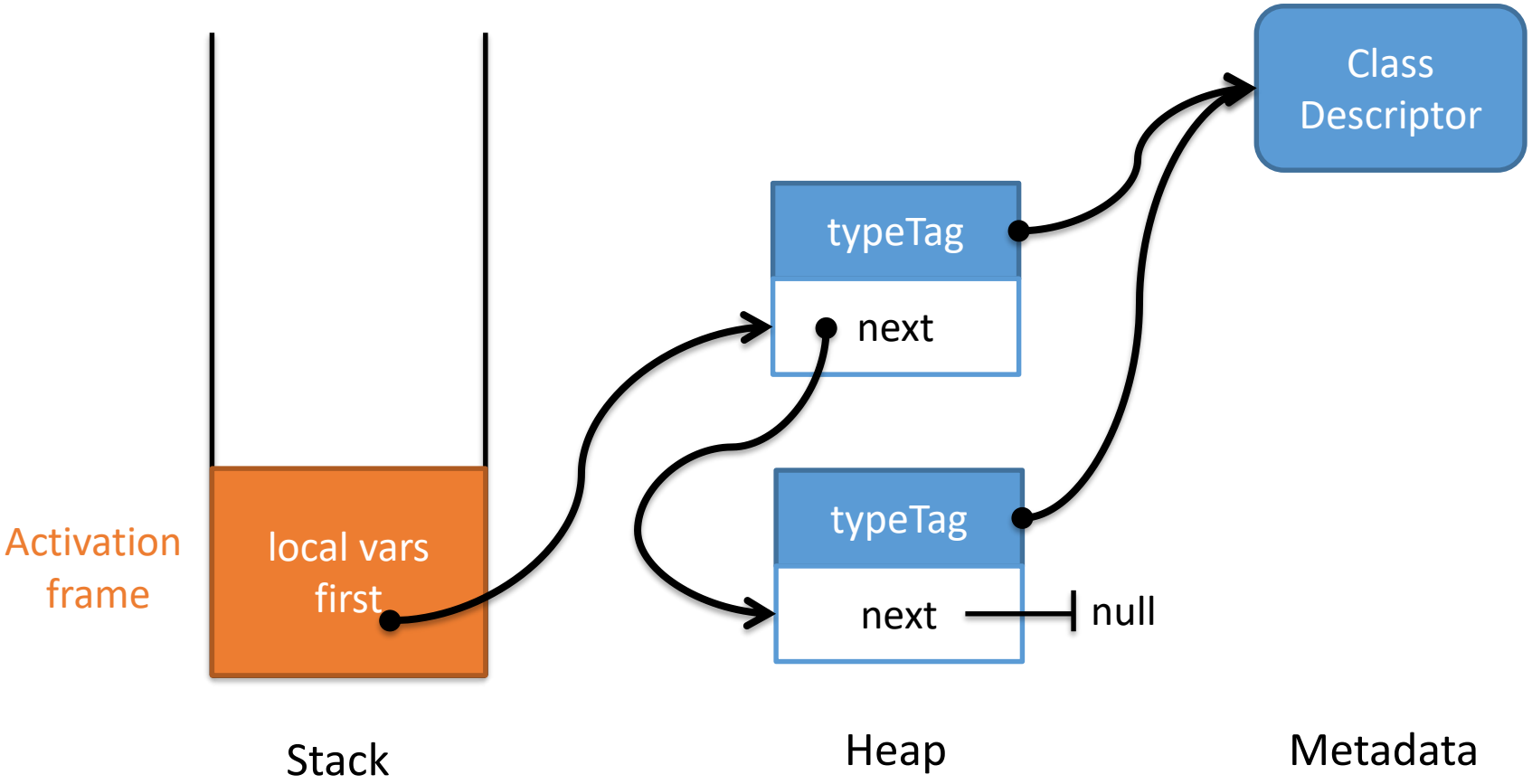
Last Lecture - Quiz

```
class Node {
   Node next;
}
```

```
Node first;
first = new Node();
first.next = new Node();
```

How does the runtime structure look like?

Runtime Structure in VM



For which purpose do we need the type tag?

Purpose of Type Tag

- Interpreter
  - Type of fields and arrays
- Runtime
  - Dynamic type casts/tests
  - Virtual method calls

Our focus today

Today's Topics

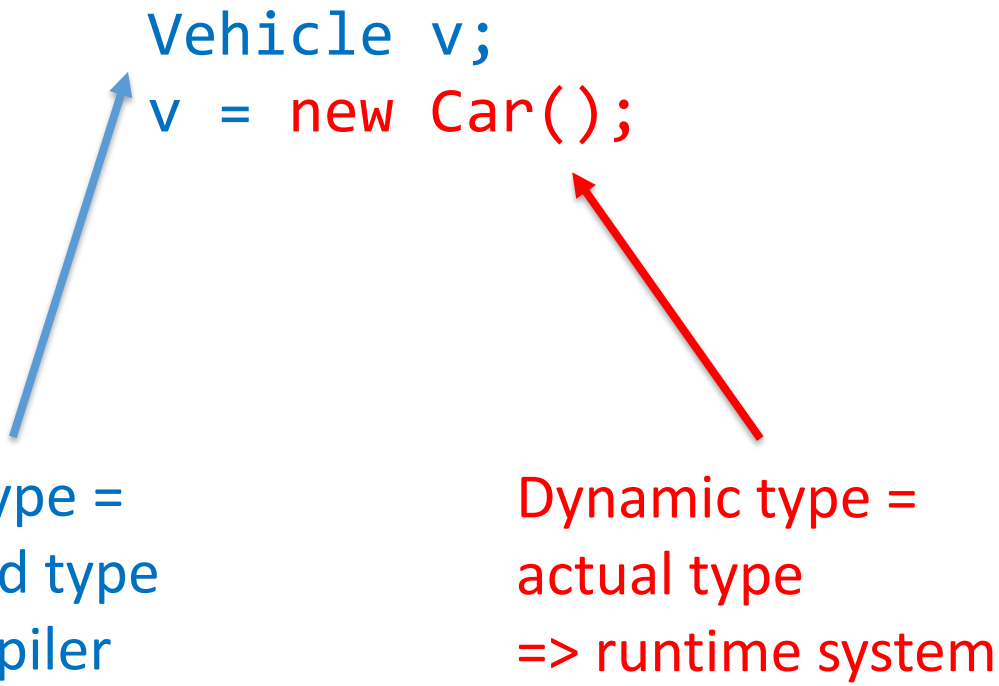
- Type polymorphism
- Type tests and casts
- Virtual method calls

Learning Goals

- Know the runtime structures for type polymorphism and dynamic dispatch
- Be able to extend the VM for OO single inheritance

Type Polymorphism

- Subtype is assignment compatible to base type



Type Test & Cast

```
Vehicle v;
Car c;
v = …;

if (v instanceof Car) {
  c = (Car)v;
}
```

Type test

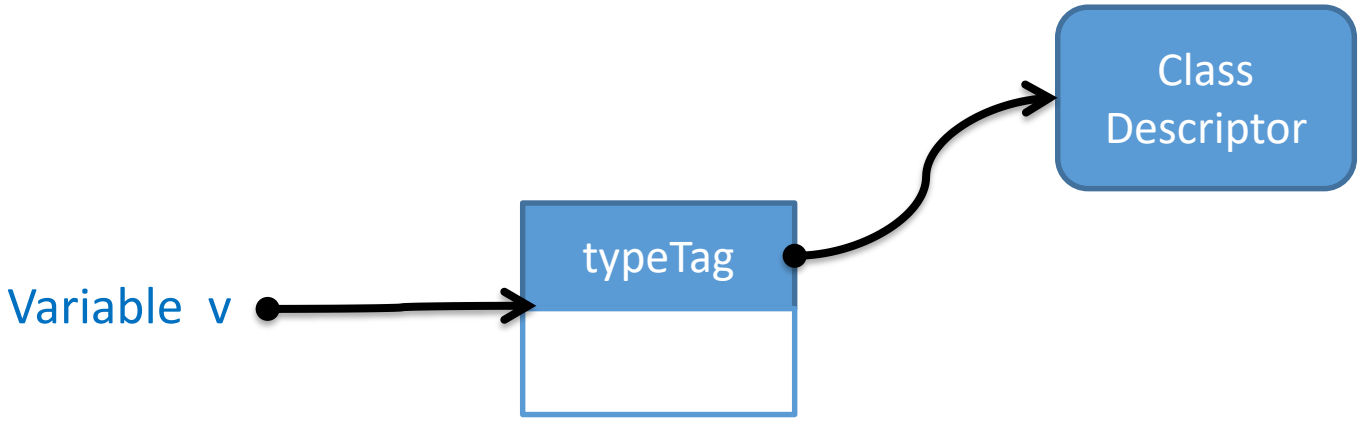
Type cast (guard)

Matter of the runtime system (in VM)

How can this be determined?

Dynamic Type Lookup

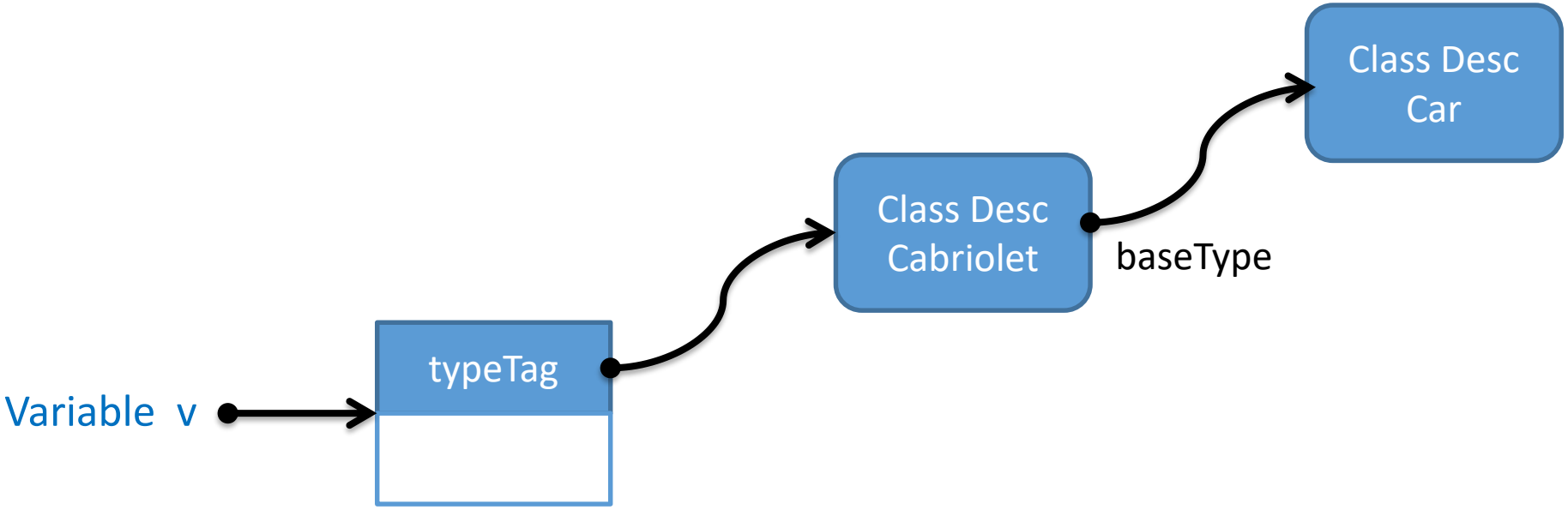
- Class descriptor = dynamic type of the object



v instanceof Car? Is it sufficient to check that the dynamic type is Car?

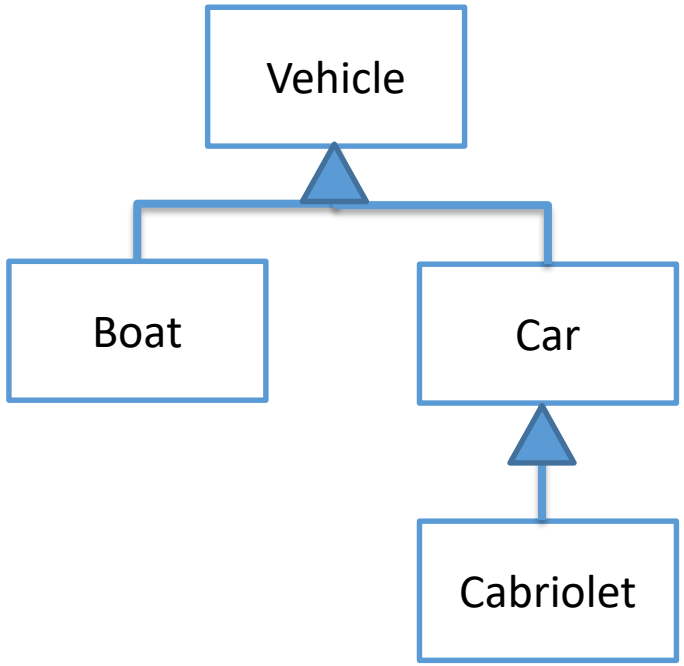
Indirect Base Type

- Need to traverse chain of base types



inefficient

Fix Levels of Inheritance

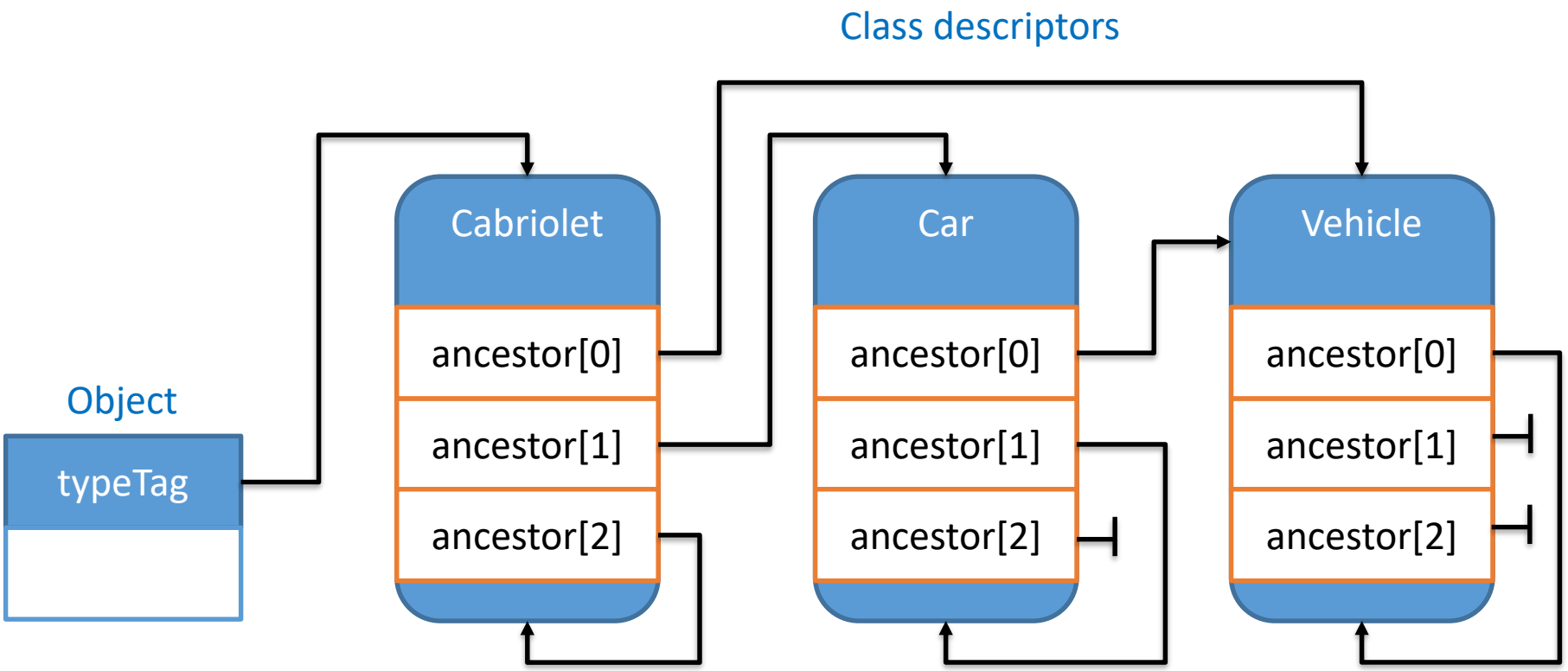


Level 0

Level 1

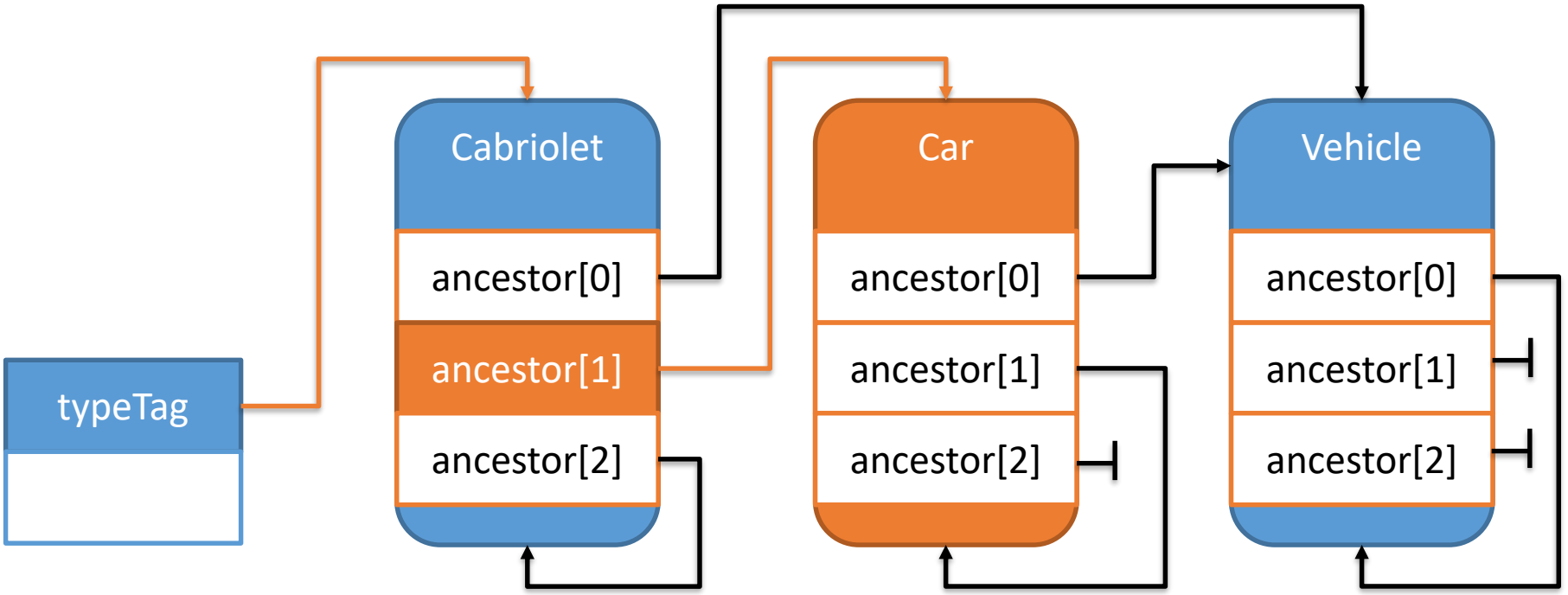
Level 2

Ancestor Table



ancestor[i] = Pointer to class descriptor of level i

v instanceof Car?



Car has level 1 => typeTag.ancestor[1] => is Car

Discussion

- Constant time type tests/casts
- Number of levels is limited
  - Alternative: dynamic-length table with bounds check
- Only works for single inheritance
- Root class not used in UCI-Java
  - Compiler can decide root class type tests/casts in Java

```
class ClassDescriptor {
   private int ancestorLevel;
   private ClassDescriptor[] ancestorTable;
   …
}
```

Unnecessary Type Tests & Casts

- Dynamic = static
  - `v instanceof Vehicle`
- Or up-cast
  - `c instanceof Vehicle`

```
Vehicle v;
Car c;
```

Compiler can decide this & remove casts

Bytecode Instructions

- Type casts do not modify the object in any regard
  - Only a runtime check that the type assumption is valid

Instruction	Meaning
`instanceof <Class>`	Type test
`checkcast <Class>`	Type cast

Interpretation Patterns

instanceof

```
var instance = pop();
var target = (ClassDescriptor)instruction.getOperand();
var desc = heap.getDescriptor(instance);
int level = target.getAncestorLevel();
push(desc.getAncestorTable()[level] == target);
```

checkcast

```
var instance = peek();
instanceof
if (!pop()) {
  throw new VMException("Invalid cast");
}
push(instance);
```

How do we need to handle the "null" case in addition?

Virtual Methods

- (Non-static) methods can be overridden

```
class Vehicle {
  void drive() { ... }
  void park() { ... }
}
```

```
class Car extends Vehicle {
  void drive() { ... }
  void gear() { ... }
}
```

Overrides Vehicle.drive()

Dynamic Dispatch

- On virtual method call, the dynamic type is decisive

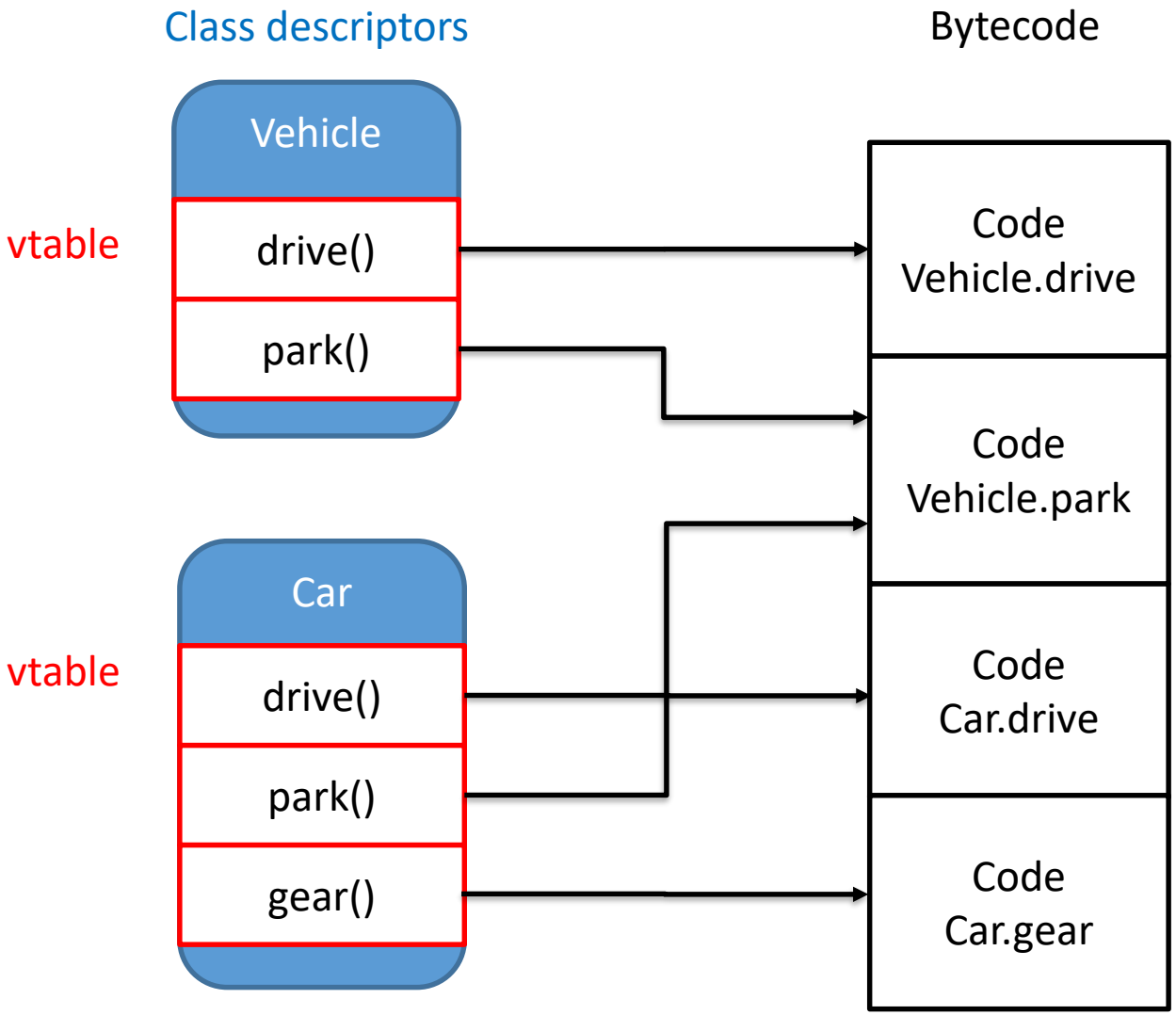
```
Car c;
c = new Car();

Vehicle v;
v = c;

v.drive();
```

Dynamic type = Car

Call of Car.drive()

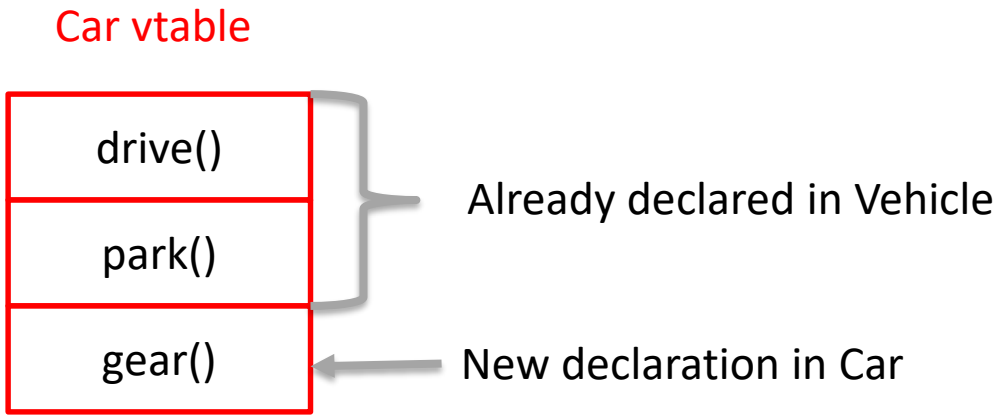
Matter of the runtime system

Virtual Method Table



Linear Extension

- Every virtual method of class has an entry
  - Methods of base class above
  - Newly declared methods in subclass below



Only works for single inheritance

Method Position

- Every virtual method has a fix position in vtable
- Position is statically known in declared type

```
class Vehicle {
  void drive() {} // pos 0
  void park() {}  // pos 1
}
```

Vehicle vtable

0	drive()
1	park()

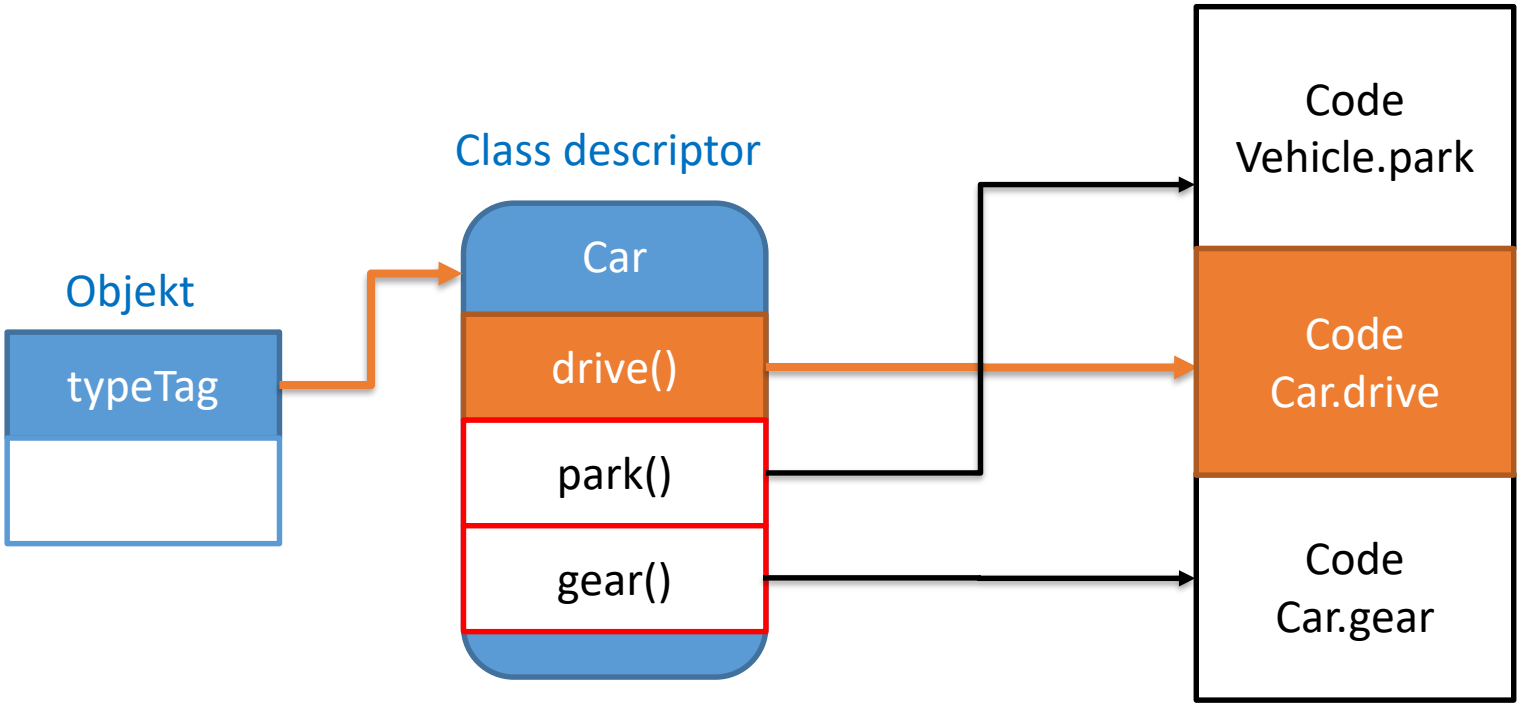
```
class Car extends Vehicle {
  void drive()   // pos 0
  void gear()    // pos 2
}
```

Car vtable

0	drive()
1	park()
2	gear()

Virtual Call

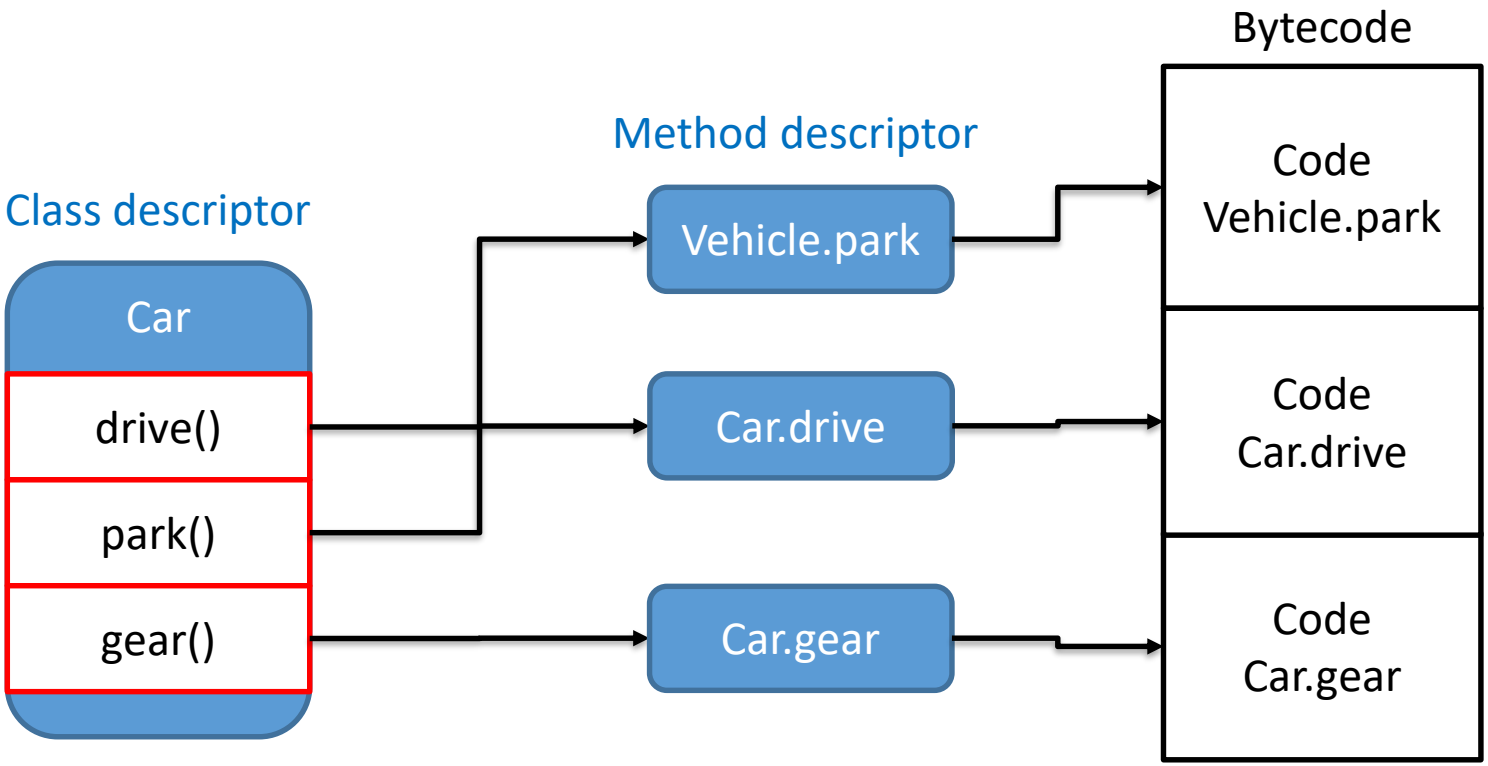
```
v.drive();
```



Branch to v.typeTag.vtable[0]

Method Descriptor

- Additional indirection via method descriptor
- Interpreter needs info about types of params/locals



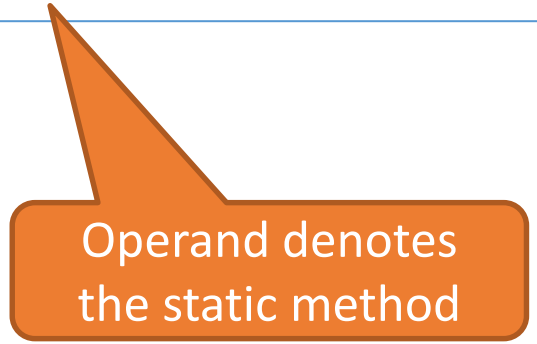
Discussion

- Efficient mechanism
  - Constant time method code lookup
- Only works like this for single inheritance
  - More tricky for multi-inheritance or interfaces
- Loader prepares vtable and position

```
class ClassDescriptor {
  private MethodDescriptor[] virtualTable;
  …
}

class MethodDescriptor {
  private int position;
  …
}
```

Bytecode Instruction

Instruction	Meaning
`invokevirtual <Method>`	Virtual Method Call



Interpretation Pattern

invokevirtual

```
… // pop arguments
var staticMethod = (MethodDescriptor)instruction.getOperand();
var target = pop();
var type = heap.getDescriptor(target);
int position = staticMethod.getPosition();
var dynamicMethod = type.getVirtualTable()[position];
… // call dynamic method
```

Advanced Type Polymorphism

In other OO languages

- Interfaces (Java, C#)
- Multi-inheritance (C++)

Many problems and more difficult implementation

More information in appendix (optional)

Review: Learning Goals

- ✓ Know the runtime structures for type polymorphism and dynamic dispatch
- ✓ Be able to extend the VM for OO single inheritance

Appendix

Self-Study (Optional)

Interface Polymorphism

- Multi-implementation of interfaces

```
interface A {
  void f();
}
```

```
interface B {
  void g();
}
```

```
class ABImpl implements A, B {
  void f() { ... }
  void g() { ... }
}
```

Interface Calls

- Method calls via interface type

```
A x = ...
B y = ...

x.f();

y.g();
```

Dynamic dispatch

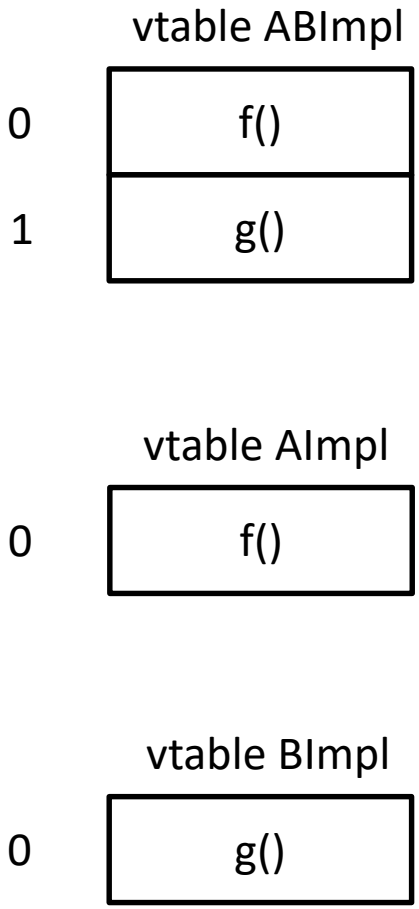
Can we use the same approach like with the vtable before?

Attempt

```
class ABImpl implements A, B {
  void f() { ... }
  void g() { ... }
}
```

```
class AImpl implements A {
  void f() { ... }
}
```

```
class BImpl implements B {
  void g() { ... }
}
```



Different positions

```
B y = ...;
y.g(); // pos ??
```

Interface Support (1)

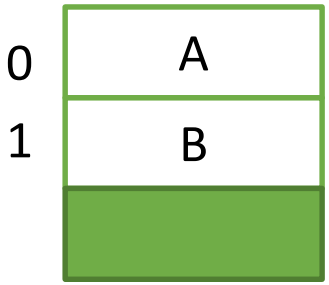
- Globally number interfaces

Interface	Number
A	0
B	1
C	2
…	…

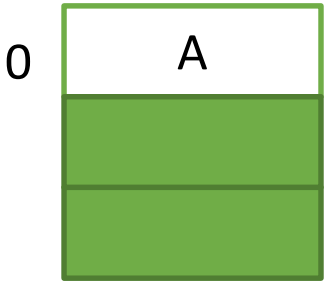
Interface Support (2)

- Assign interface table (itable) per class descriptor

```
class ABImpl implements A, B
```

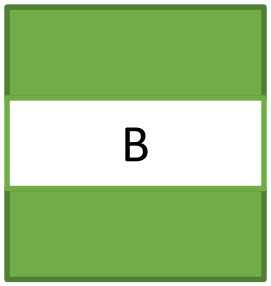



```
class AImpl implements A
```



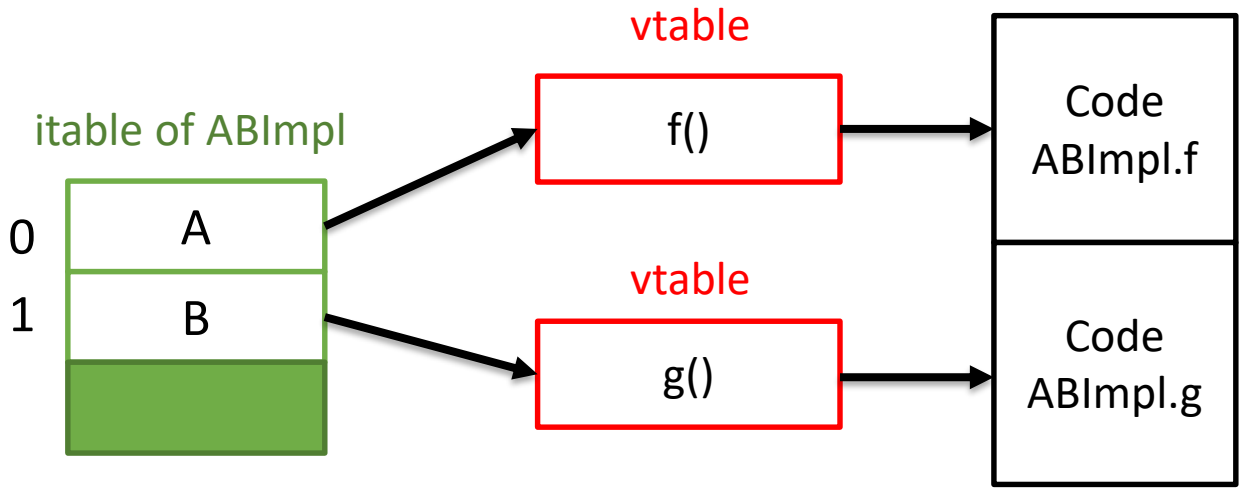
Insert interfaces according to their number = position

```
class BImpl implements B
```



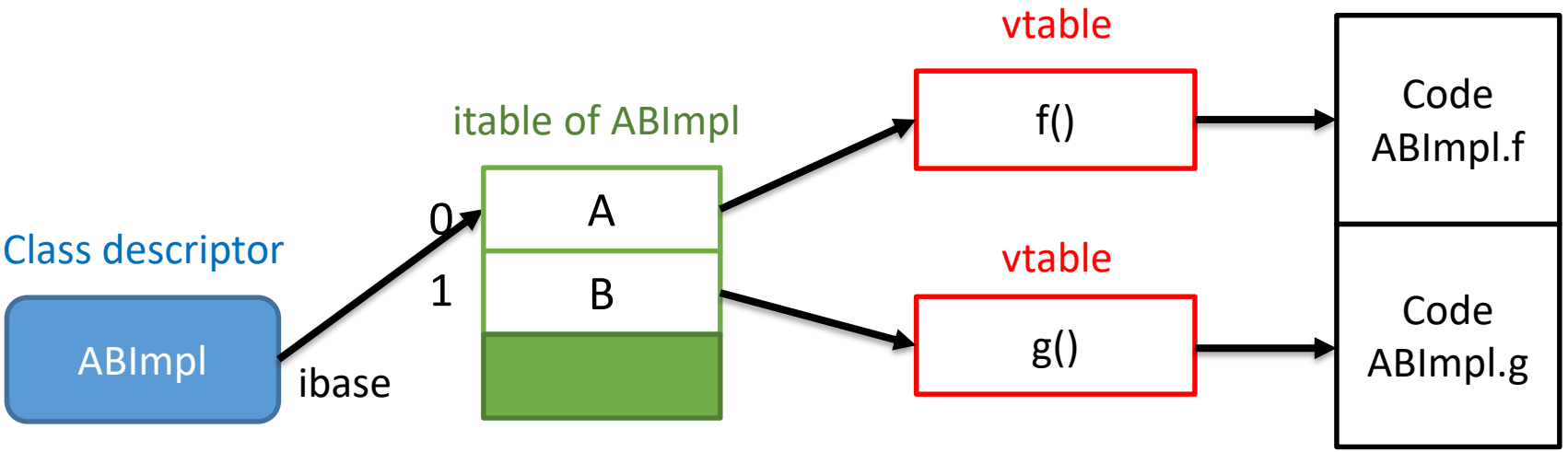
Interface Support (3)

- Entries in itable point to vtable



Interface Support (4)

- Class descriptor points to itable (ibase-pointer)

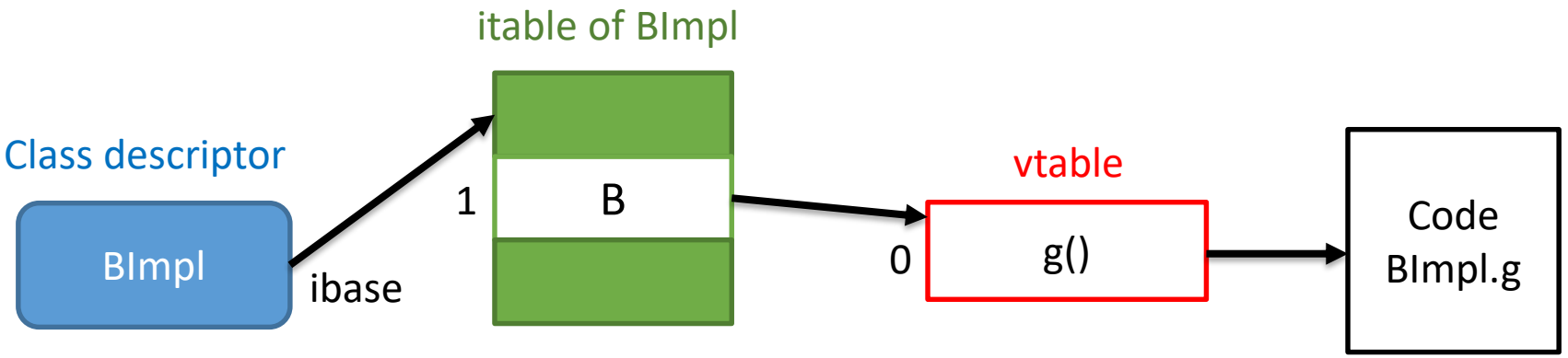


Interface Call

Statically known

```
B y = ...;
y.g();
```

Interface number 1
Method position 0



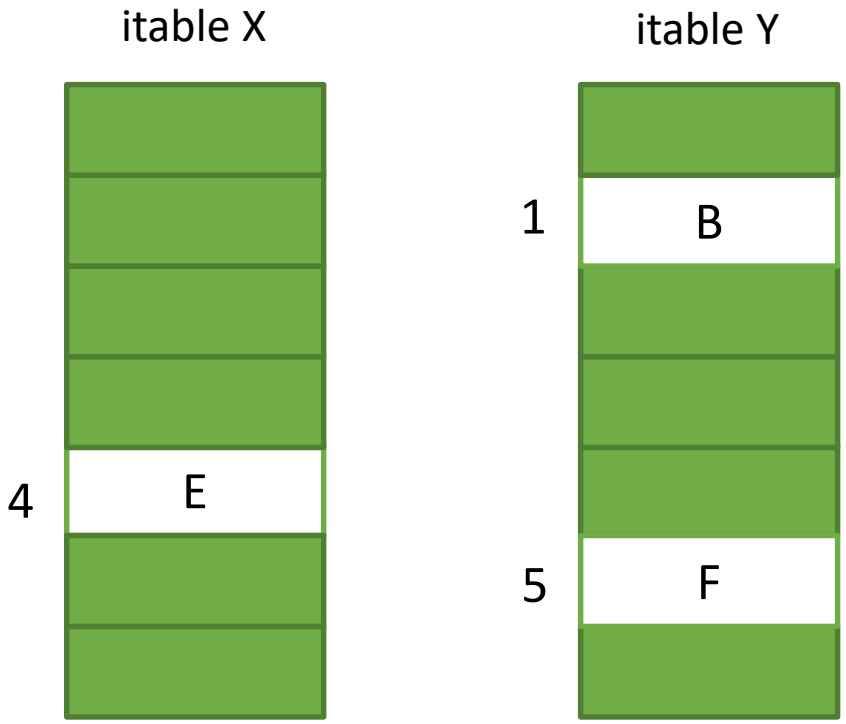
Branch to y.typeTag.ibase[1][0]

Interface Test and Cast

- Type test: `y instanceof A`
- Type cast: `(A)y`
- Same mechanism
  - Check whether interface entry exists in itable
  - `y.typeTag.ibase[1] != null`

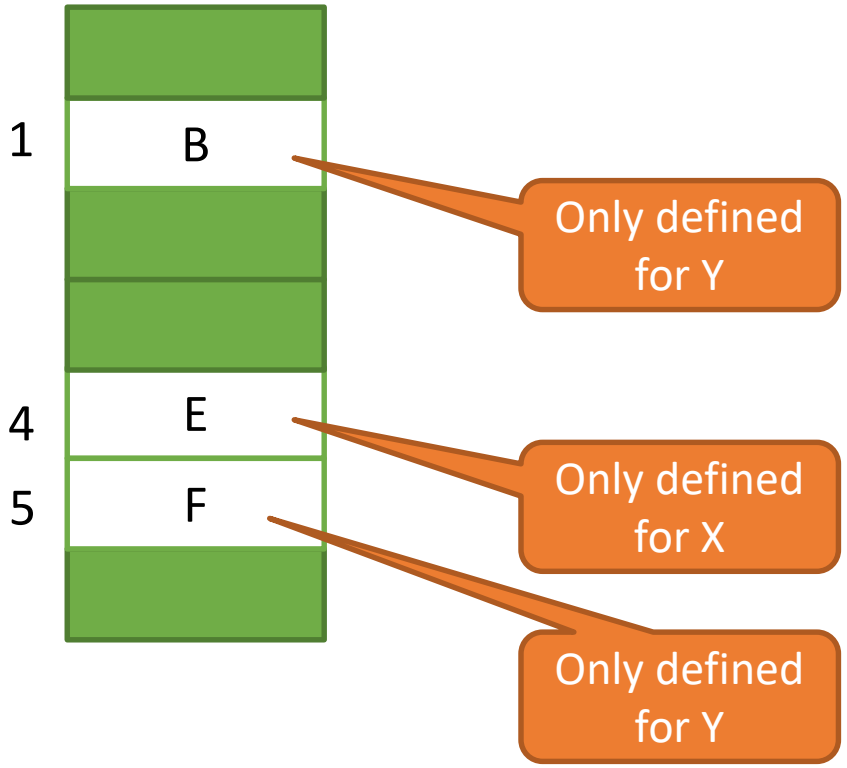
Memory Problem

- Large itables in the case of many interfaces
- Many holes because of global numbering



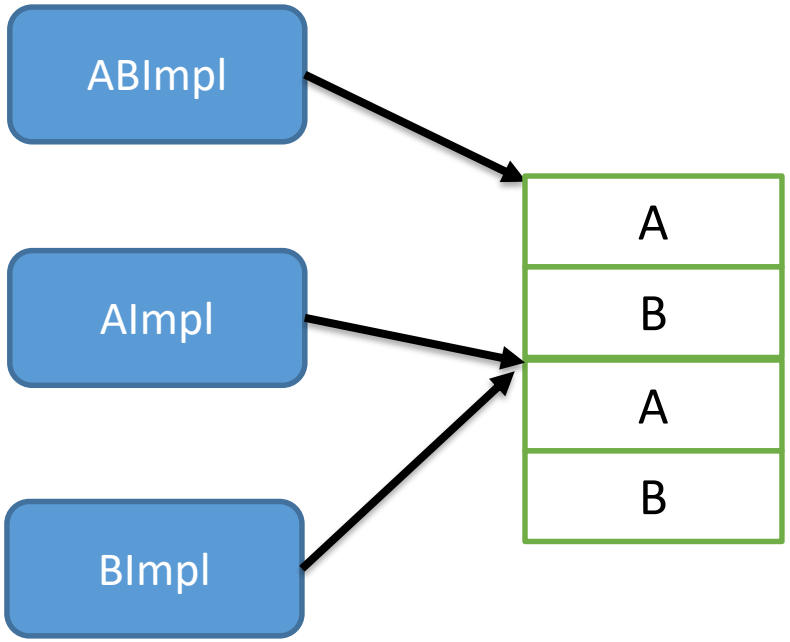
Interleaved Interface Table

- Overlap itables in memory (without clash)
- Check, if entry applies to specific type
  - Record class descriptor in vtable

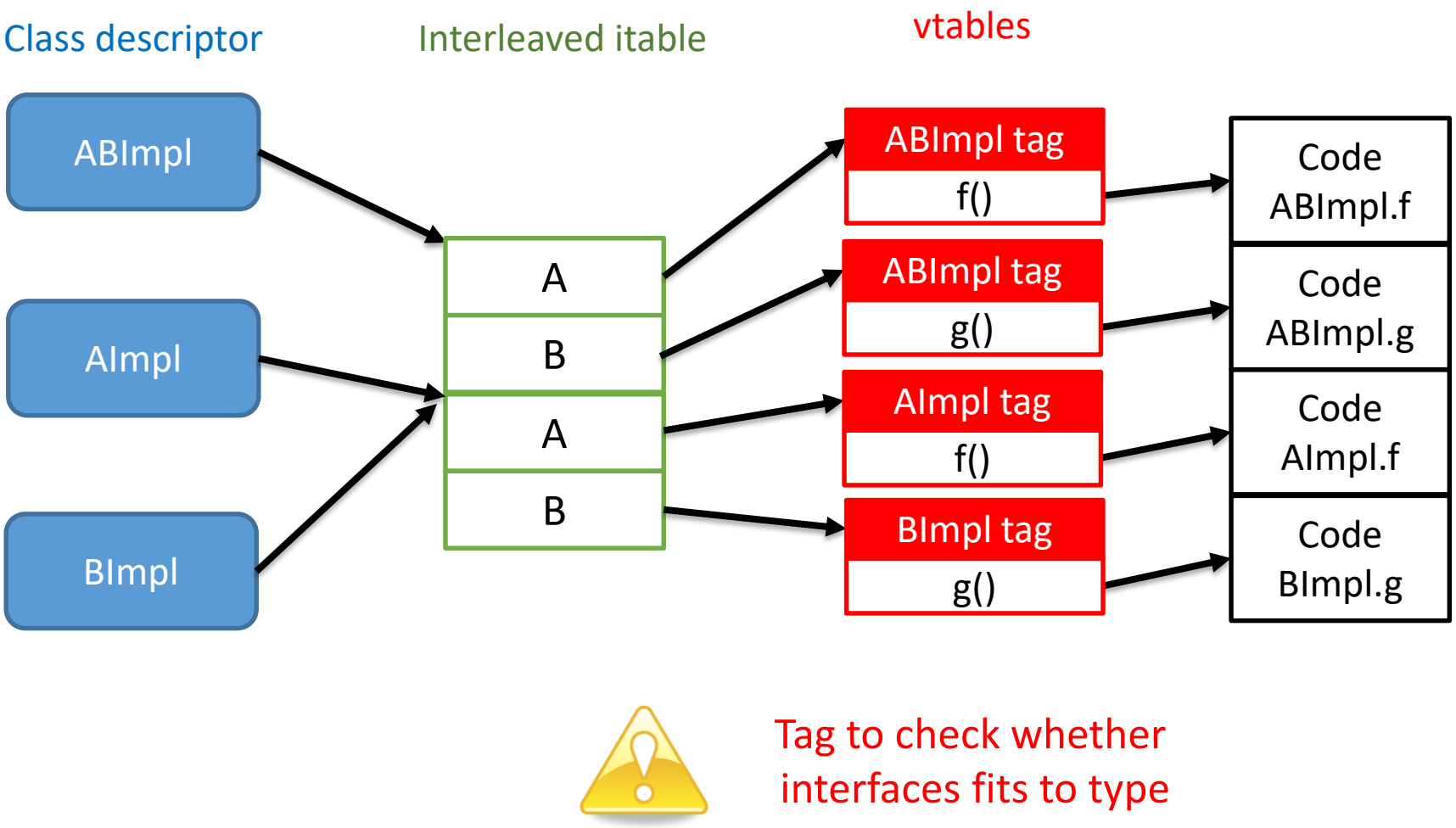


Different Offsets

- Interleaving is also possible with different offsets



Full Picture: Interface Support



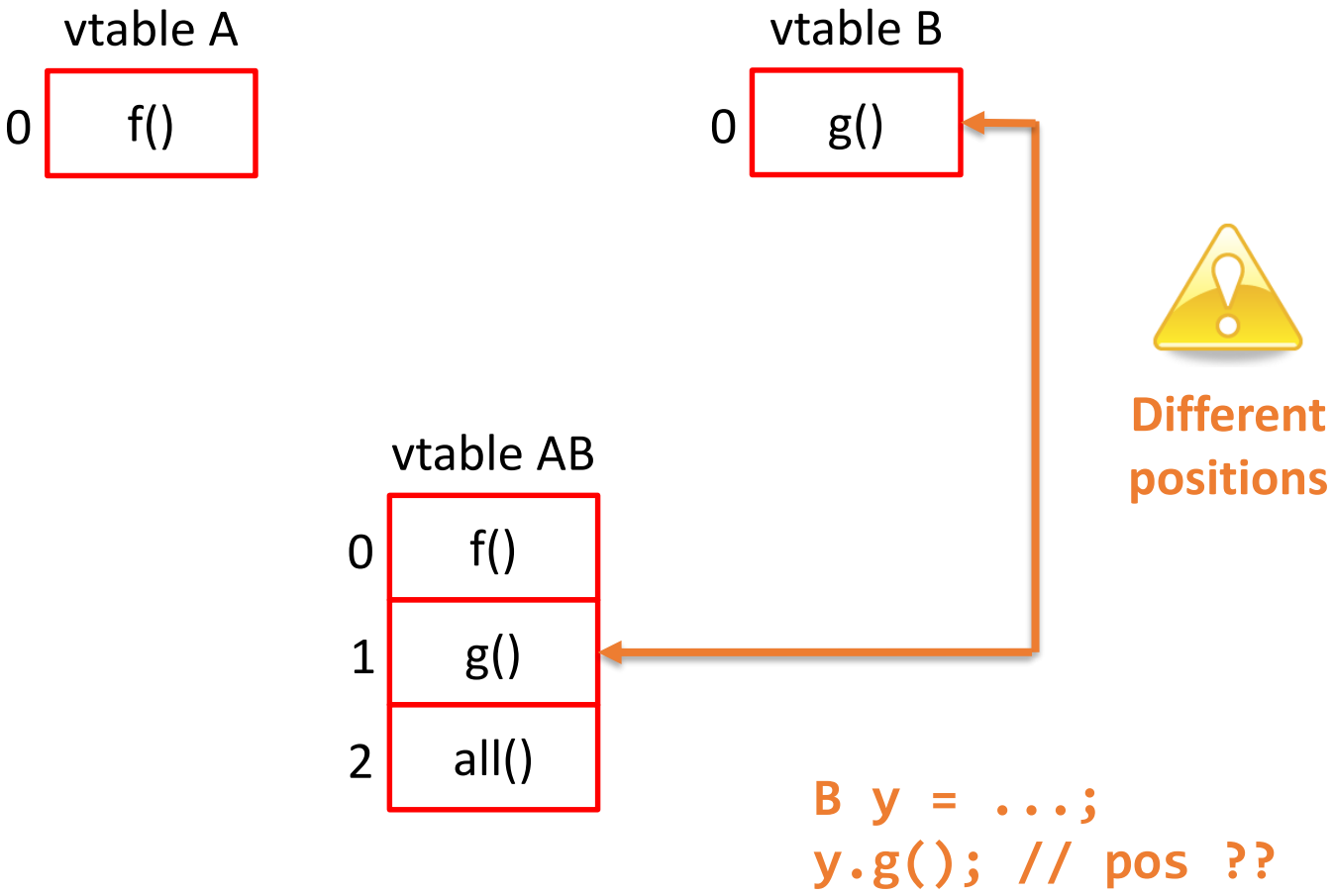
Multi-Inheritance

```
class A {
  int a;
  void f() { a = … }
}
```

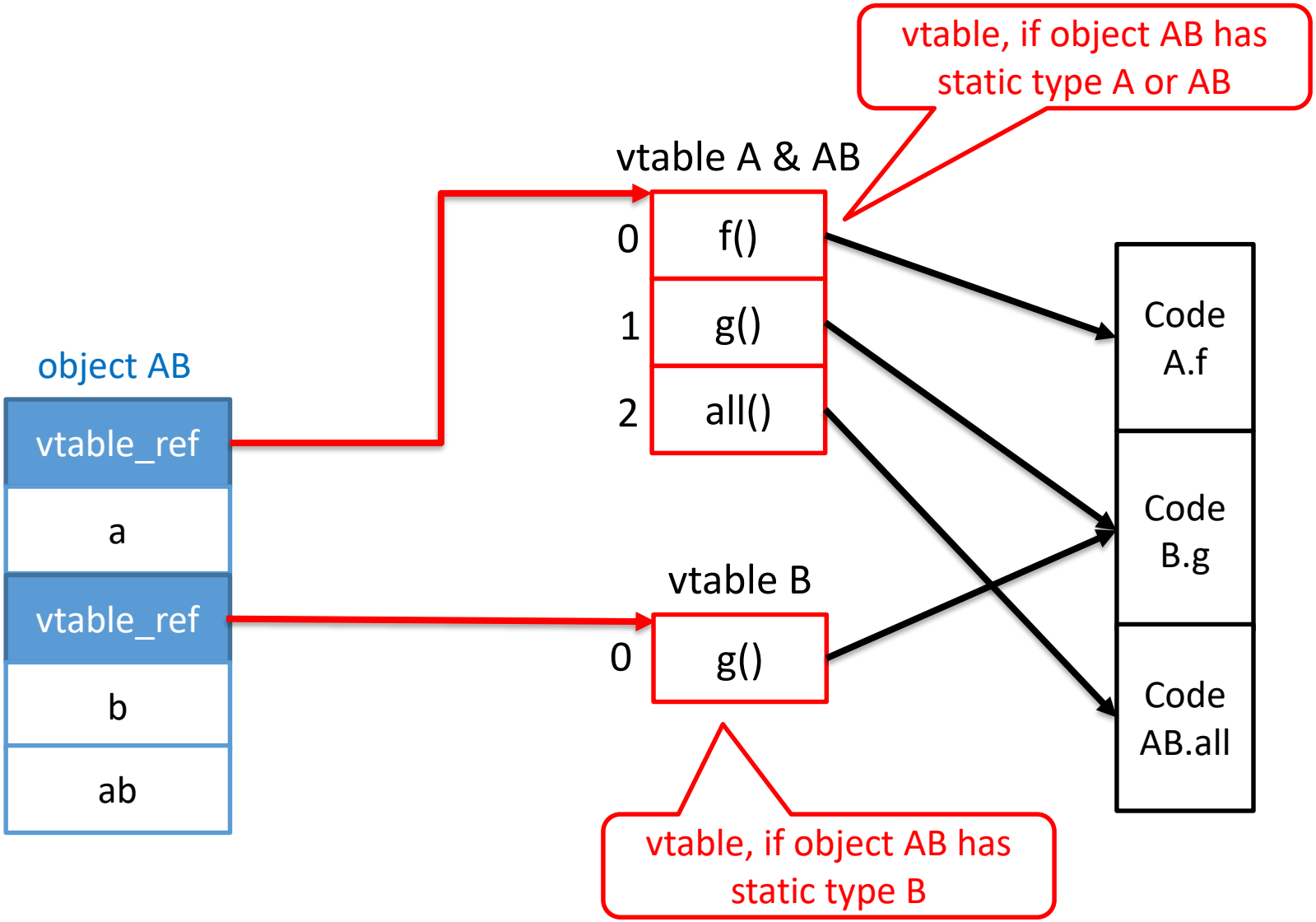
```
class B {
  int b;
  void g() { b = … }
}
```

```
class AB : A, B {
  int ab;
  void all() {
    a = ab; b = ab;
  }
}
```

Virtual Table Problem

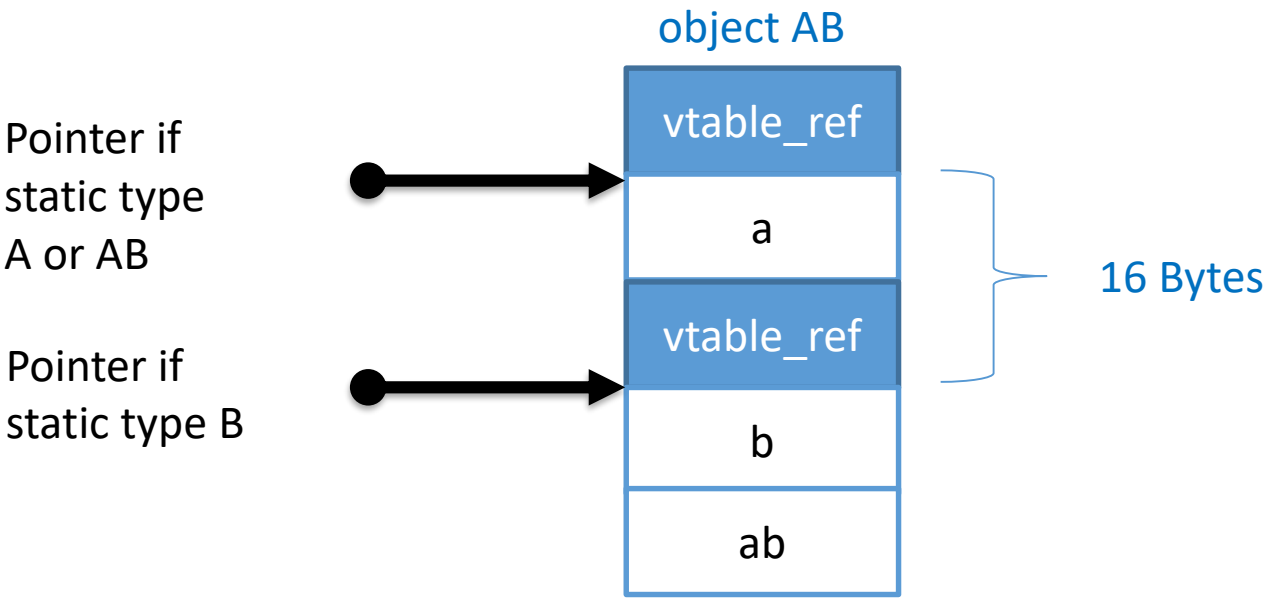


Solution for Virtual Table



Multi-Inheritance Virtual Tables

- Subclass points to multiple virtual tables
  - Depending on static type
- Pointer to object differs depending on static type



Pointer Shifts on Type Casts

Up-cast

```
AB x = …;
B y = x;
```

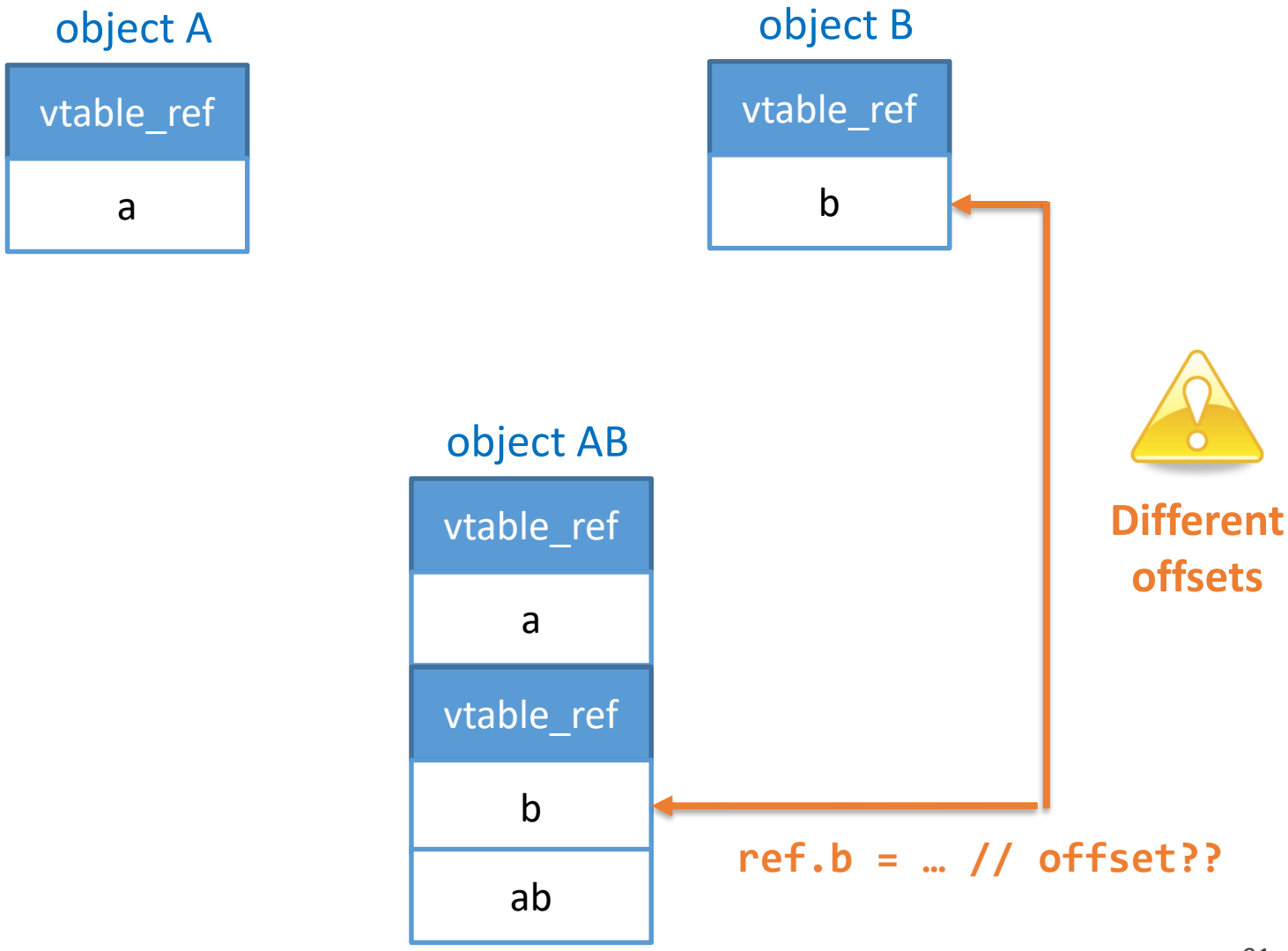
y = x + 16

Down-cast

```
B y = …;
AB x = (AB)y;
```

x = y - 16

Field Offset Problem



Solution: Shift of «this»-Reference

```
B y = new B();
y.g();
```

this = y

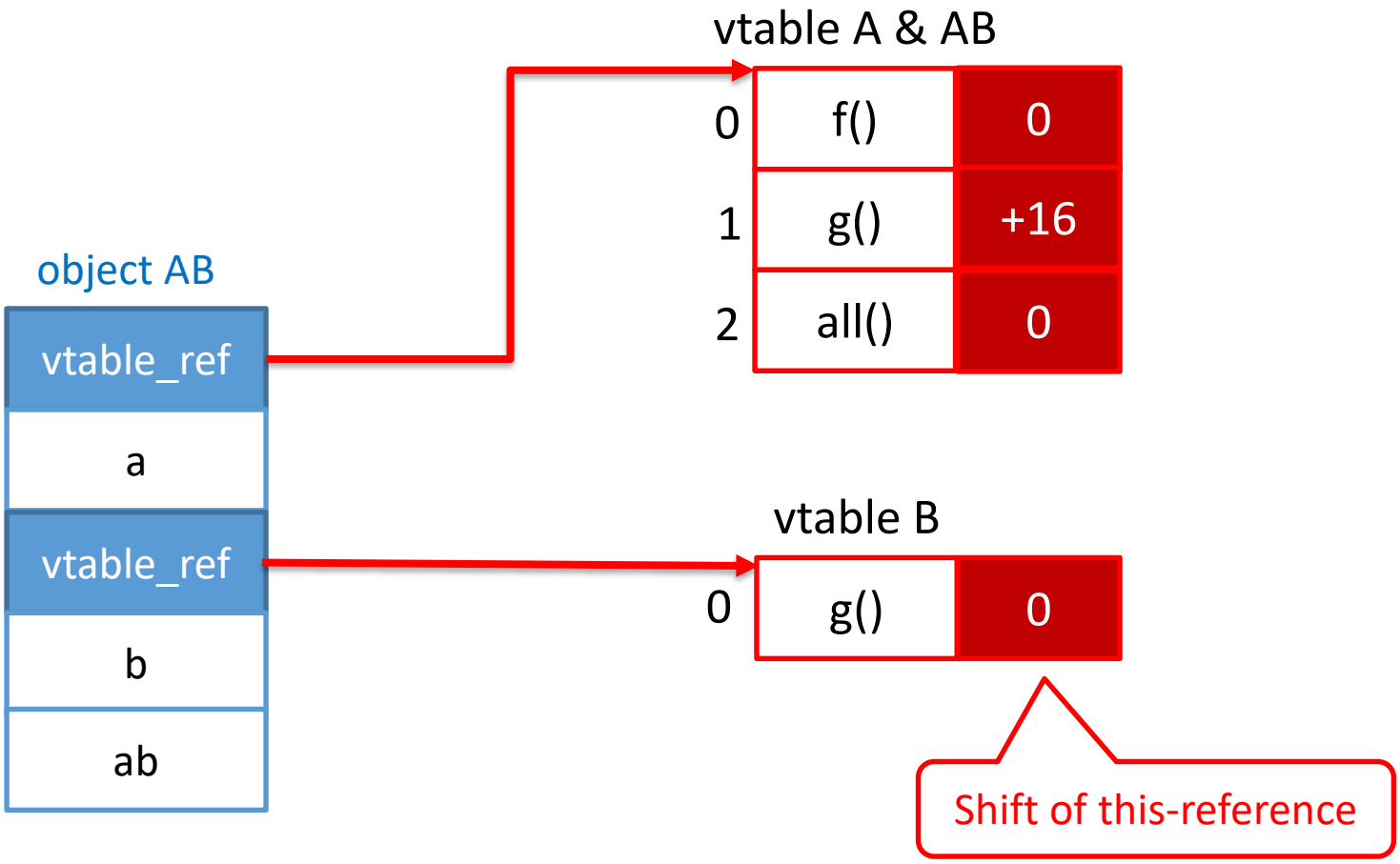
```
AB x = new AB();
x.g();
```

this = x + 16

```
void g() { b = … }
```

b: Offset 0 to this
(from perspective of class B)

this-Shifts in Virtual Table



Multi-Inheritance: Other Problems

- Conflicts
  - Method with same signature inherited multiple times
  - Resolution: Overriding, selection
- Diamant problem
  - Variable inherited multiple times (over different paths)
  - Resolution: Duplicate or share