PA5: Last Hints on Symbol Table: Other Info to Keep

For each VarSTE:

is it a local variable or a member variable?

For each class: what will the object size be?

For each method:

- a VarSTE for "this" parameter
- how many parameters including the implicit "this" parameter?
- how many local variables
- mangled method name, e.g. classname_method

PA5: Last Type Checking Hints

Possible Type Errors in PA5:

- AssignStatement LHS and RHS
- Checking the return value – already done
- Checking type of receiver in a method call – see Tuesday's slides

Generating Code for Classes

Generating Code
for member variables

outIdExp:

1) Lookup id in symbol table to get VarSTE

2) If the VarSTE is a member variable:

2a) Look up VarSTE for "this" and generate code
that loads the value of "this" into registers r31:r30.

3) load variable into a register(s) using the base+offset store in the VarSTE.

4) Push the variable value on the stack.

Generating Code for an Assignment Statement

outAssignStatement:

1) Lookup id in symbol table to get VarSTE

2) If the VarSTE is a member variable

2a) Look up VarSTE for "this" and generate code that loads the value of "this" into registers r31:r30.

3) store value of expression on top of run-time stack into base+offset for VarSTE

Review: Object Memory Layout with Inheritance

Consider the following OOP code:

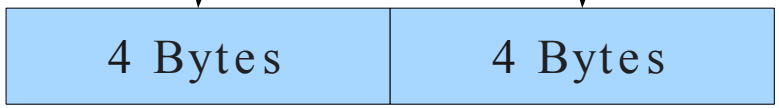
```
class Base {
    int x;
    int y;
}
class Derived extends Base {
    int z;
}
```

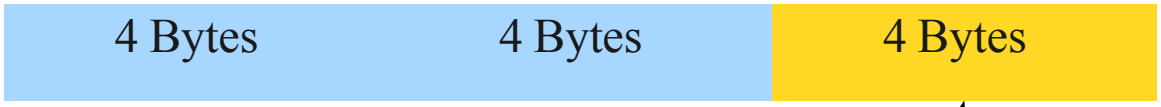
What will ***Derived*** *look like in memory?*

Review: Object Memory Layout with Inheritance

Assuming 4 bytes per int in this architecture

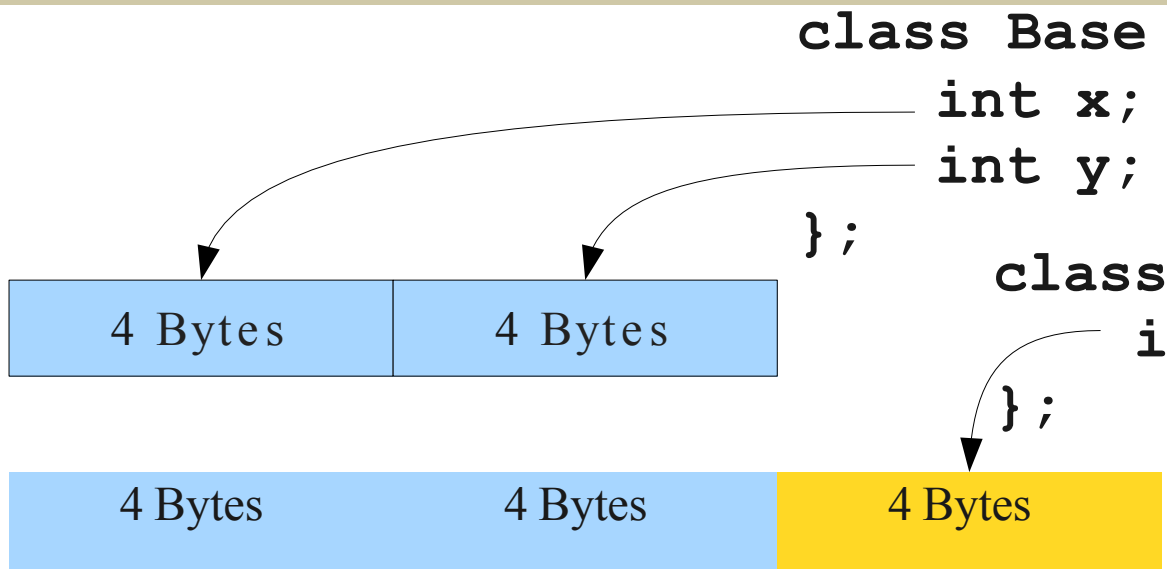
```
class Base {
    int x;
    int y;
};
```






```
class Derived extends Base {
    int z;
};
```

Member Lookup With Inheritance




```
Base ms = new Base;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms

Base ms = new Derived;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms
```

Share your Object Layout Examples

Single Inheritance Object Memory Layout Summary

- Derived class layout after Base class layout, etc.

Rationale: A pointer of type B pointing at a D object still sees the B object at the beginning.

Operations done on a D object through the B reference **guaranteed to be safe; no need to check what B points at dynamically.**

What About Methods and Method calls (with inheritance)?

- Methods are mostly like regular functions, but with two complications:

- How do we know what receiver object to use?

- How do we know which function to call at runtime (dynamic dispatch)?

Implementing **this**

Receiver.memberfunction(actual parameters)

- Inside a member function, the name **this** refers to the current receiver object.
- This information (pun intended) needs to be communicated into the function.
- **Idea:** Treat **this** as an implicit first parameter.
- Every n-argument member function is really an (n+1)-argument member function whose first parameter is the **this** pointer.

this is Clever

```
class MyClass {
    int x;
    void myFunction(int arg) {
        this.x = arg;
    }
}

MyClass m = new MyClass;
m.myFunction(137);
```

this is Clever

```
class MyClass {
    int x;
}
void MyClass_myFunction(MyClass this, int arg){
    this.x = arg;
}

MyClass m = new MyClass;
m.myFunction(137);
```

`this` is Clever

```
class MyClass {
    int x;
}
void MyClass_myFunction(MyClass this, int arg){
    this.x = arg;
}

MyClass m = new MyClass;
MyClass_myFunction(m, 137);
```

The mangled function name

`this` Rules

- **When generating code to call a member function:** pass an object as the `this` parameter representing the receiver object.

- **Inside member function:** treat `this` as just another parameter to the member function.

- **When implicitly referring to a field of `this`:** use this extra parameter as the object in which the field should be looked up.

Generating Code for Method Calls

Need to pass in the receiver reference as the first parameter.

Receiver.memberfunction(actual parameters)

outCallExp

1) Look up the ClassSTE from the receiver type.
   Then lookup the MethodSTE from the ClassSTE scope.
2) Generate code that pops parameters off the stack and
   into the appropriate registers from right to left.
   Don't forget the "this" parameter.
3) Generate code that **calls the mangled method name**.
4) Generate code that pushes the return value back on the stack

outThisExp

1) Push the value of the "this" parameter onto the run-time stack
   The "this" parameter is stored in r31:r30.

Implementing Dynamic Dispatch

Dynamic dispatch means calling a function at runtime based on the dynamic type of an object, rather than its static type.

Question:

How do we set up our runtime environment so that we can efficiently support this?

An Initial Idea

- At compile-time, get a list of every defined class.
- To compile a dynamic dispatch, emit IR code for the following logic:

```
if (the object has type A)
    call A's version of the function
else if (the object has type B)
    call B's version of the function
…
else if (the object has type N)
    call N's version of the function.
```

Analyzing our Approach

What are the problems with this strategy?

Slow!

Number of checks is O(*C*), where *C* is the number of classes the dispatch might refer to.

Gets slower the more classes there are.

It's infeasible in most languages.

What if we link across multiple source files?

What if we support dynamic class loading?

Introducing Dispatch Tables

A **dispatch table** (or **vtable**) –

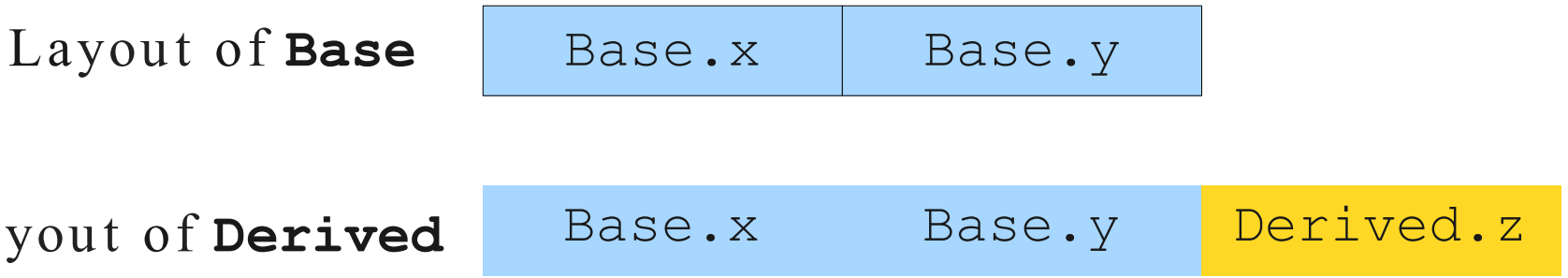
array of pointers to each member function's code for a particular class.

To invoke a member function:

1. Determine (statically) its index in the dispatch table.
2. Follow pointer at that index in the object's dispatch to the code for the function.
3. Invoke that function.

An Observation

- When laying out fields in an object, we gave every field an offset.
- Derived classes have the base class fields in the same order at the beginning.

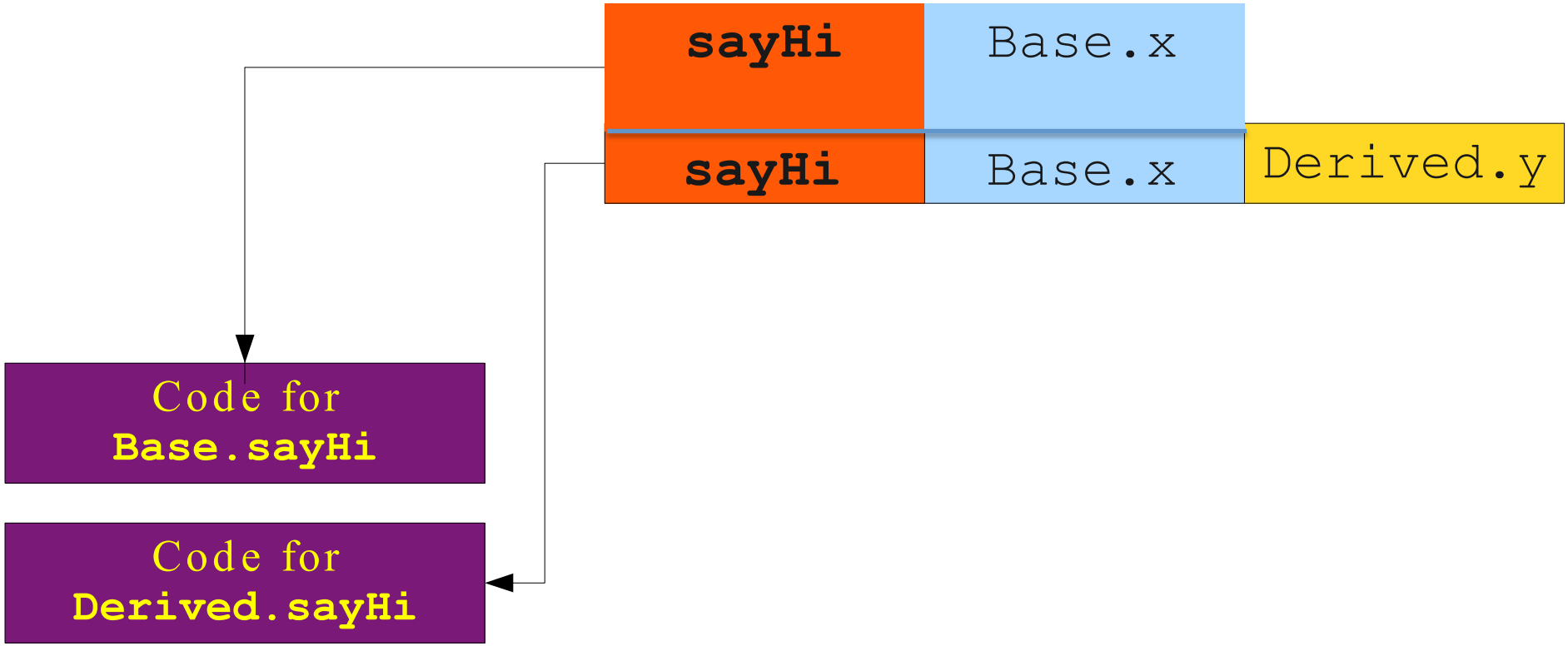


Can we do something similar with functions?

Dispatch Tables

```
class Base {
     int x;
     void sayHi() {
          Print("Base");
     }
}
```

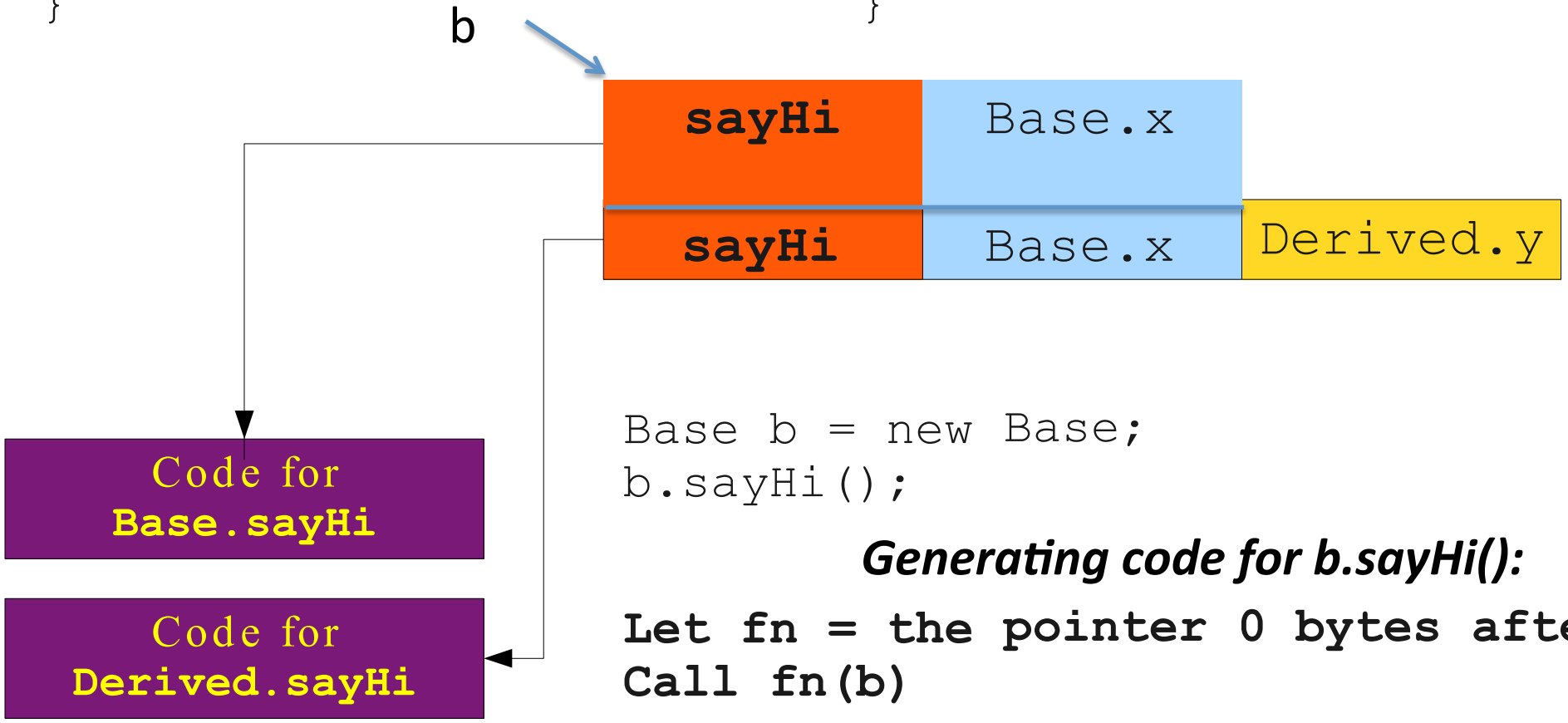
```
class Derived extends Base {
     int y;
     void sayHi() {
          Print("Derived");
     }
}
```



Dispatch Tables

```
class Base {
     int x;
     void sayHi() {
          Print("Base");
     }
}
```

```
class Derived extends Base {
     int y;
     void sayHi() {
          Print("Derived");
     }
}
```




```
Base b = new Base;
b.sayHi();
```

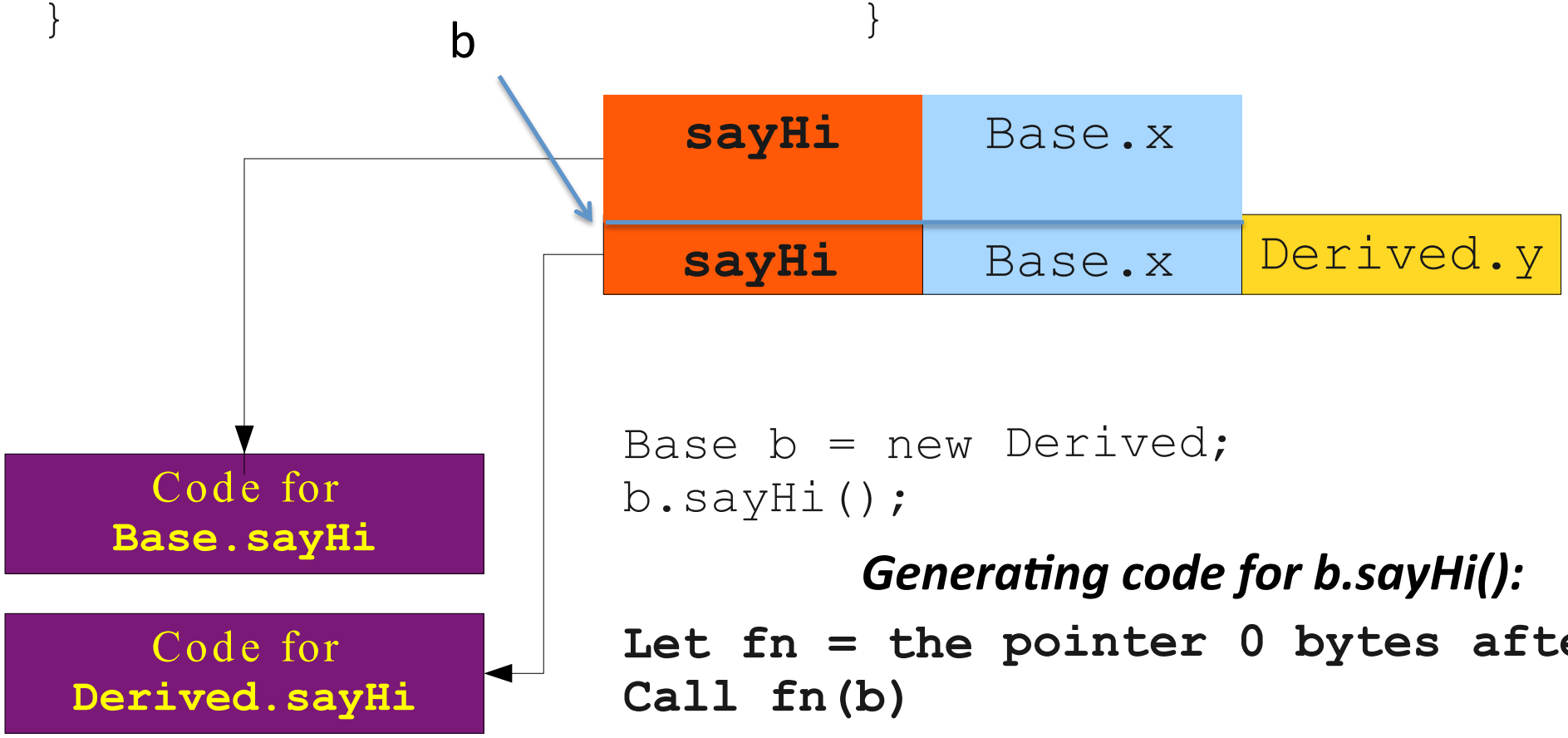
Generating code for b.sayHi():

Let fn = the pointer 0 bytes after b
Call fn(b)

Dispatch Tables

```
class Base {
     int x;
     void sayHi() {
          Print("Base");
     }
}
```

```
class Derived extends Base {
     int y;
     void sayHi() {
          Print("Derived");
     }
}
```




```
Base b = new Derived;
b.sayHi();
```

Generating code for b.sayHi():

Let fn = the pointer 0 bytes after b
Call fn(b)

More on Dispatch Tables

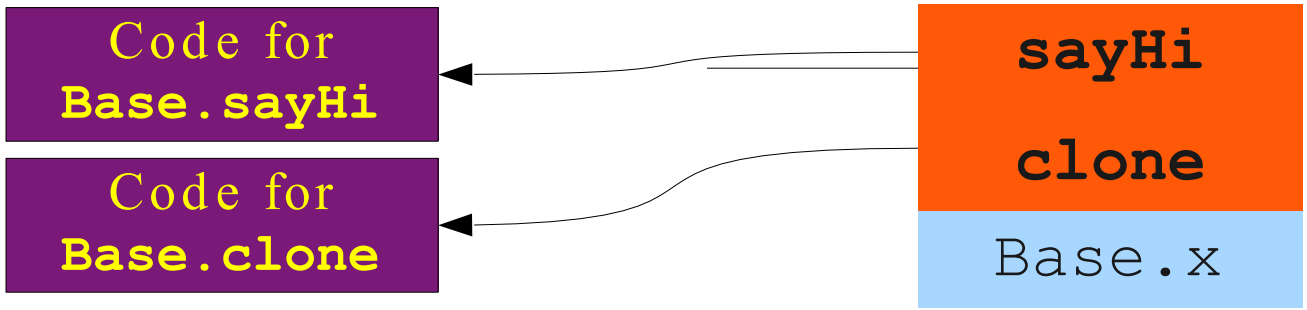
```
class Base {
    int x;
    void sayHi() {
        Print("Hi Mom!");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base
   { int y;



   Derived clone() {
       return new Derived;
   }
}
```

More on Dispatch Tables

```
class Base {
    int x;
    void sayHi() {
        Print("Hi Mom!");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base
   { int y;


   Derived clone() {
       return new Derived;
   }
}
```

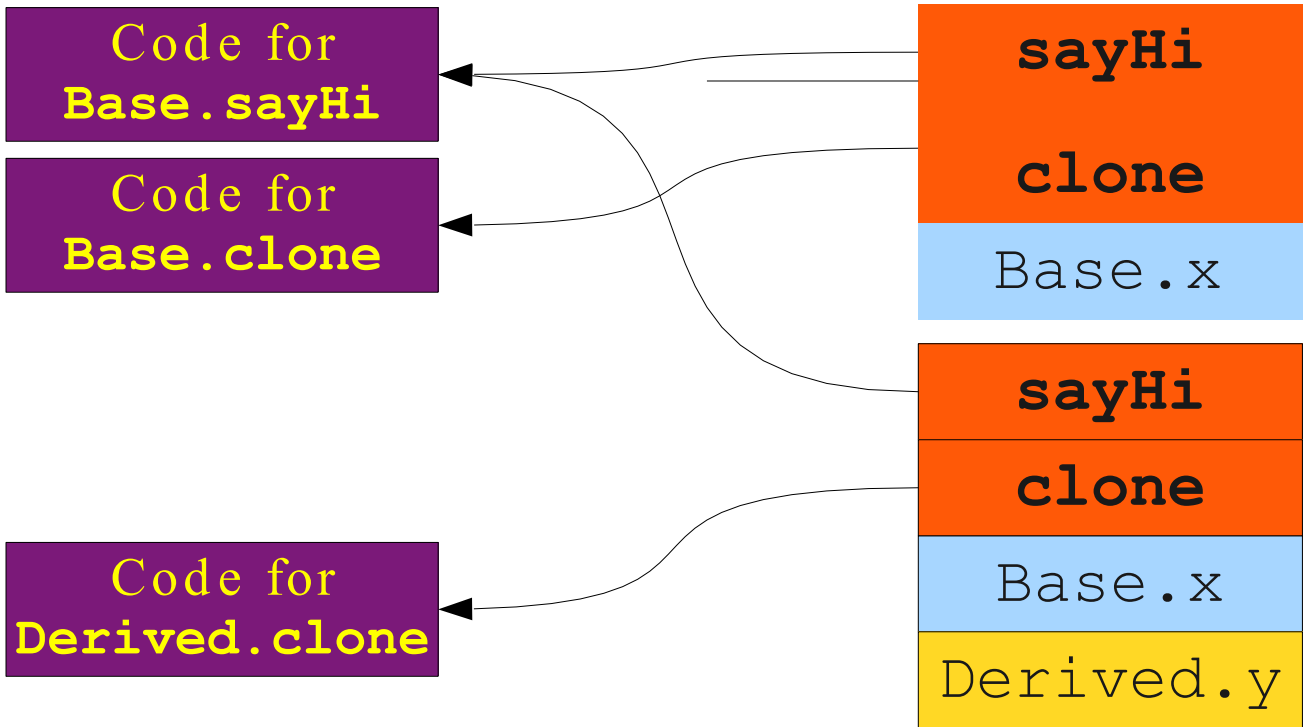


Code for
Derived.clone

More on Dispatch Tables

```
class Base {
    int x;
    void sayHi() {
        Print("Hi Mom!");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base
   { int y;


    Derived clone() {
        return new Derived;
    }
}
```



Analyzing our Approach

Advantages

Time to determine function to call is O(1).

(and a good O(1) too!)

Disadvantages

Object sizes are ?

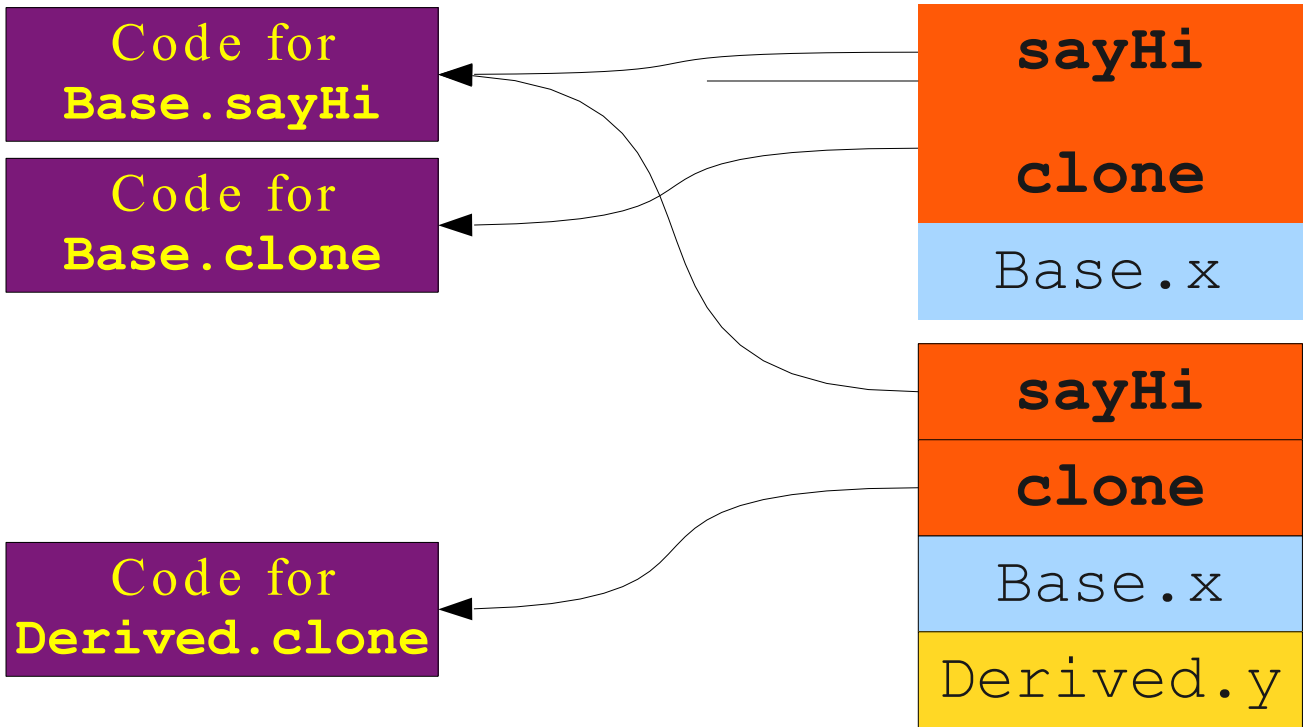
Each object needs to have space for ?.

Object creation is slower. Why?

A Common Optimization: From this…

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

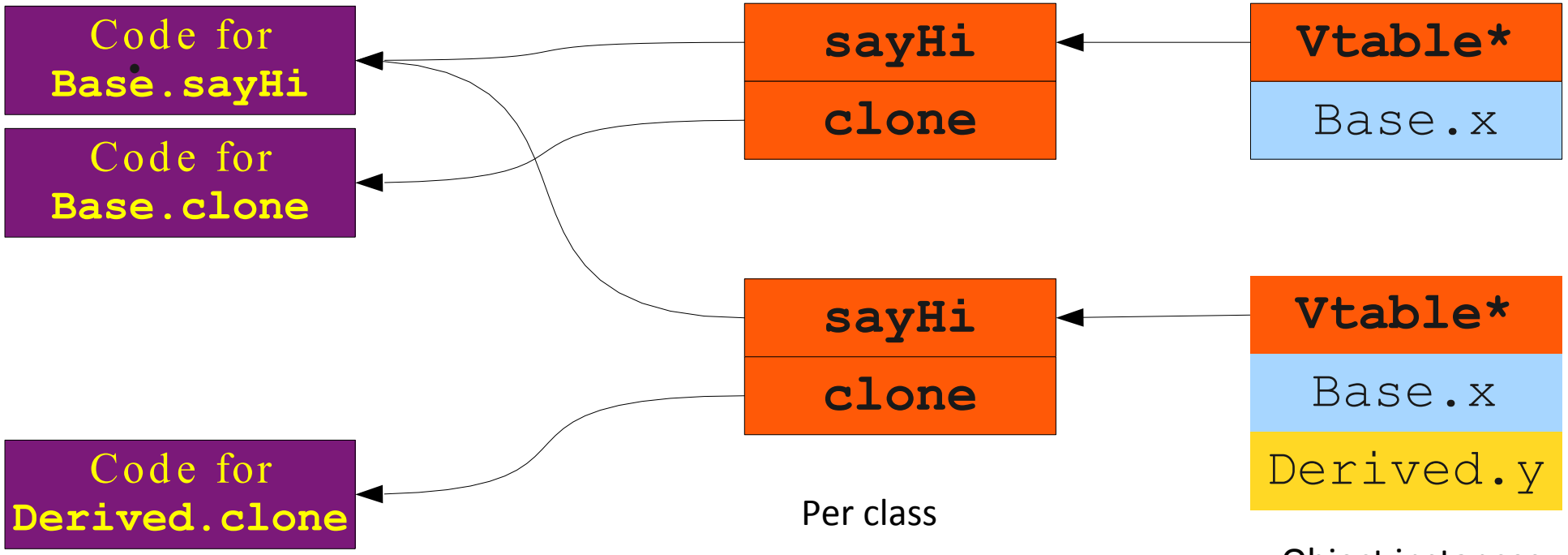
```
class Derived extends Base
   { int y;


   Derived clone() {
       return new Derived;
   }
}
```



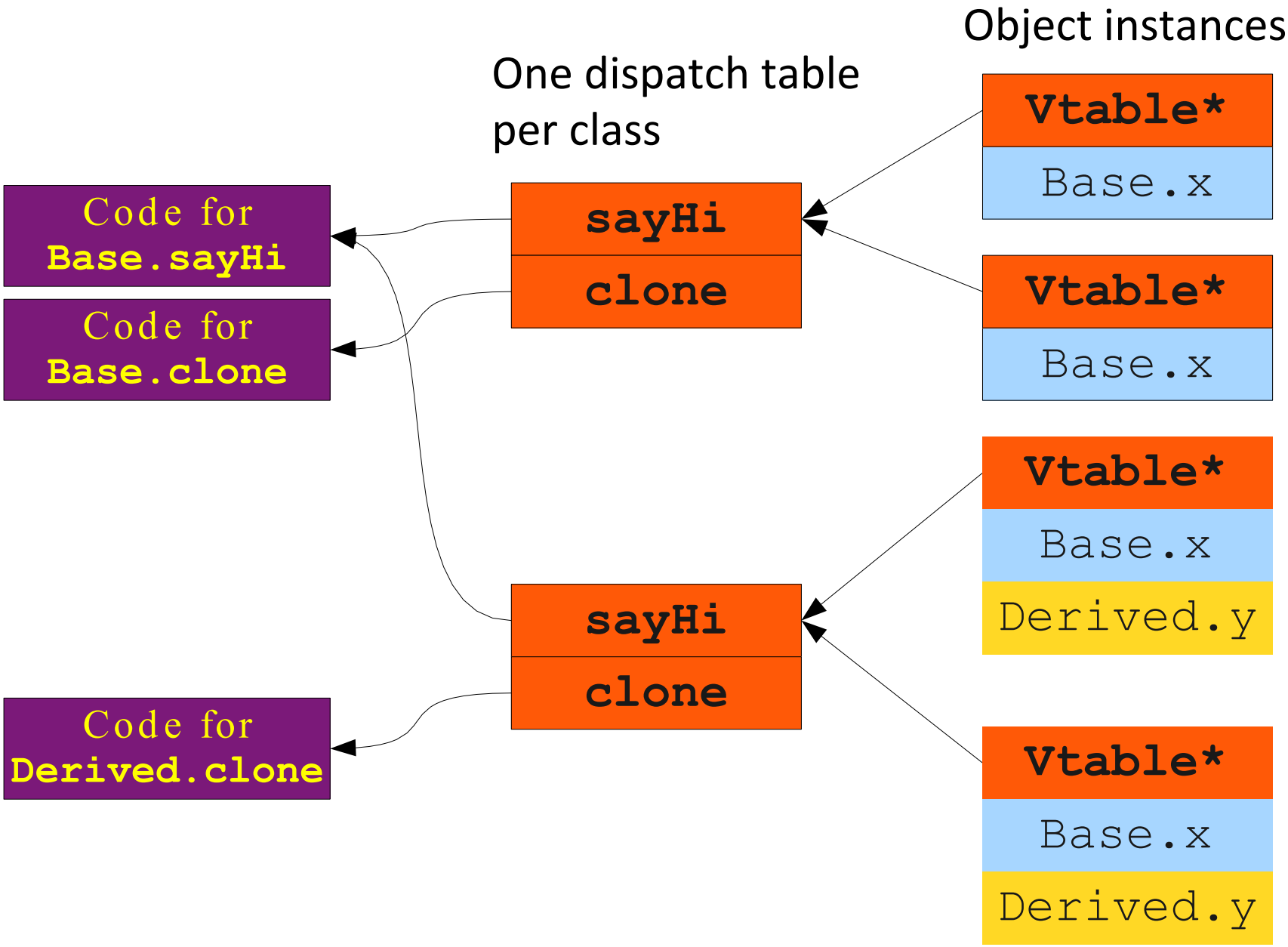
A Common Optimization: To This...

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

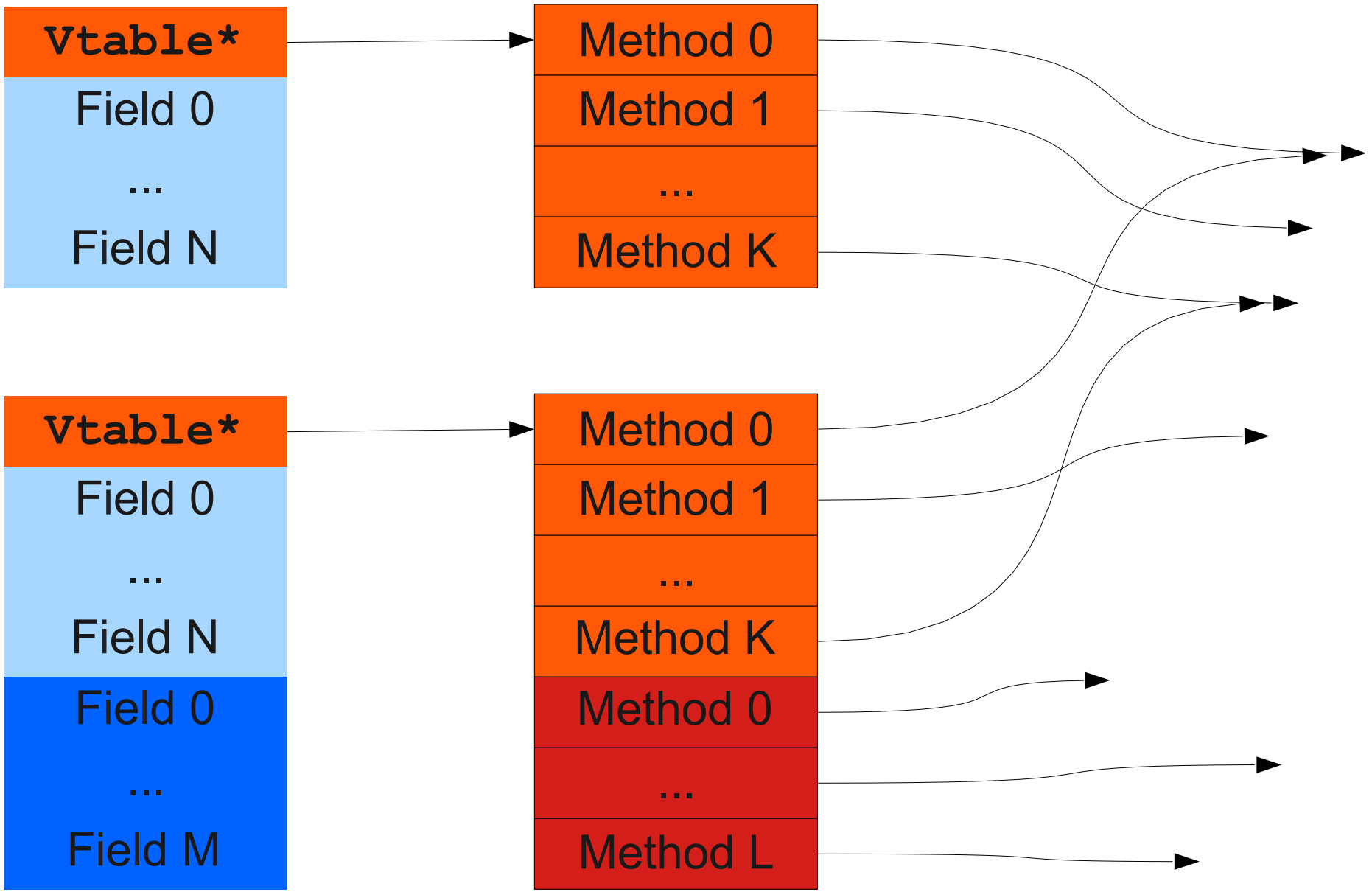
```
class Derived extends Base
   { int y;


   Derived clone() {
       return new Derived;
   }
}
```



Thus, objects in Memory



Generalized Object Layout



Summary: Dynamic Dispatch in O(1)

- Create a single instance dispatch table for each class.
- Each object stores a pointer to the dispatch table.
- Can follow the pointer to the table in O(1).
- Can index into the table in O(1).
- Can set the dispatch table pointer of new object in O(1).
- Increases the size of each object by O(1).

This is the solution used in most C++ and Java implementations.

Your Turn to Draw Pictures

- Draw the objects, dispatch tables

```
Class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int
     {a <- a+d};
};
Class B extends A {
  b: Int <- 2;
  f(): Int { a } ;
  g(): Int { a<- a – b};
};
```

```
Class C extends B {
   c: Int <- 3;
   h(): Int {a <- a * c};
};
```

Suppose the following ops occur:

```
A x = new A();
B y = new B();
A z = new A();
C w = new C();
C t = new C();
```