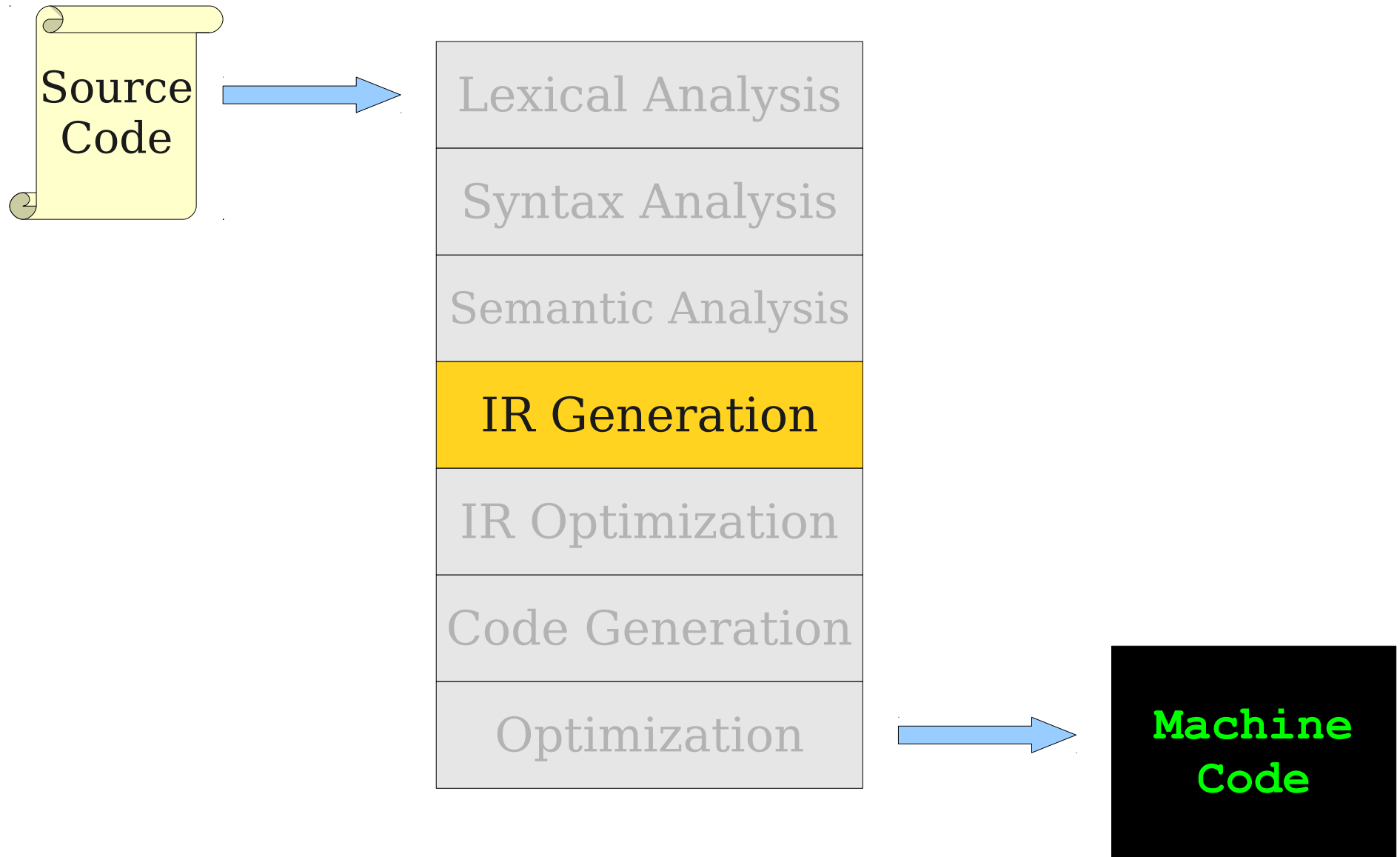


Runtime Environment

Part II

May 8th 2013

Where We Are



Review

- IR Generation: why
- Data representations
 - primitive types: byte, char, float
 - arrays: one-dimensional, multi-dimensional
- Encoding functions
 - activation record (stack frame)
 - activation tree
 - runtime stack
 - calling conventions

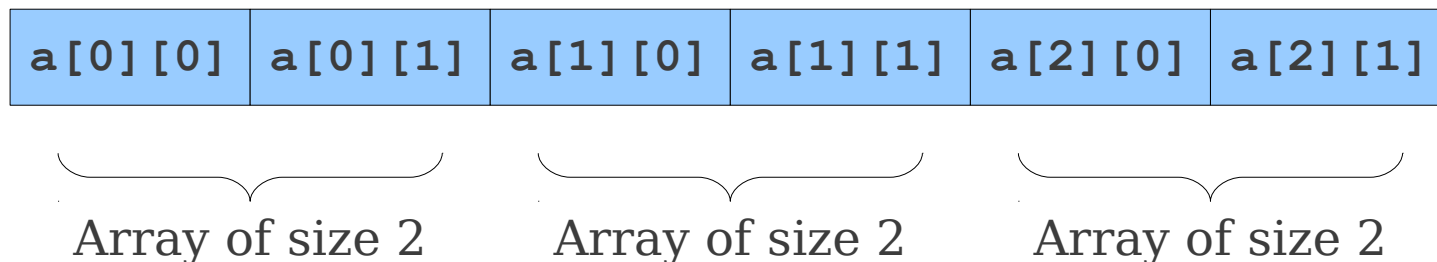
Why Do IR Generation?

- **Simplify certain optimizations.**
 - Machine code has many constraints that inhibit optimization. (Such as?)
 - Working with an intermediate language makes optimizations easier and clearer.
- **Have many front-ends into a single back-end.**
 - `gcc` can handle C, C++, Java, Fortran, Ada, and many other languages.
 - Each front-end translates source to the GENERIC language.
- **Have many back-ends from a single front-end.**
 - Do most optimization on intermediate representation before emitting code targeted at a single machine.

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

```
int a[3][2];
```



Review: The Stack

- Function calls are often implemented using a stack of **activation records** (or **stack frames**).
- Calling a function pushes a new activation record onto the stack.
- Returning from a function pops the current activation record from the stack.
- Questions:
 - **Why** does this work?
 - Does this **always** work?

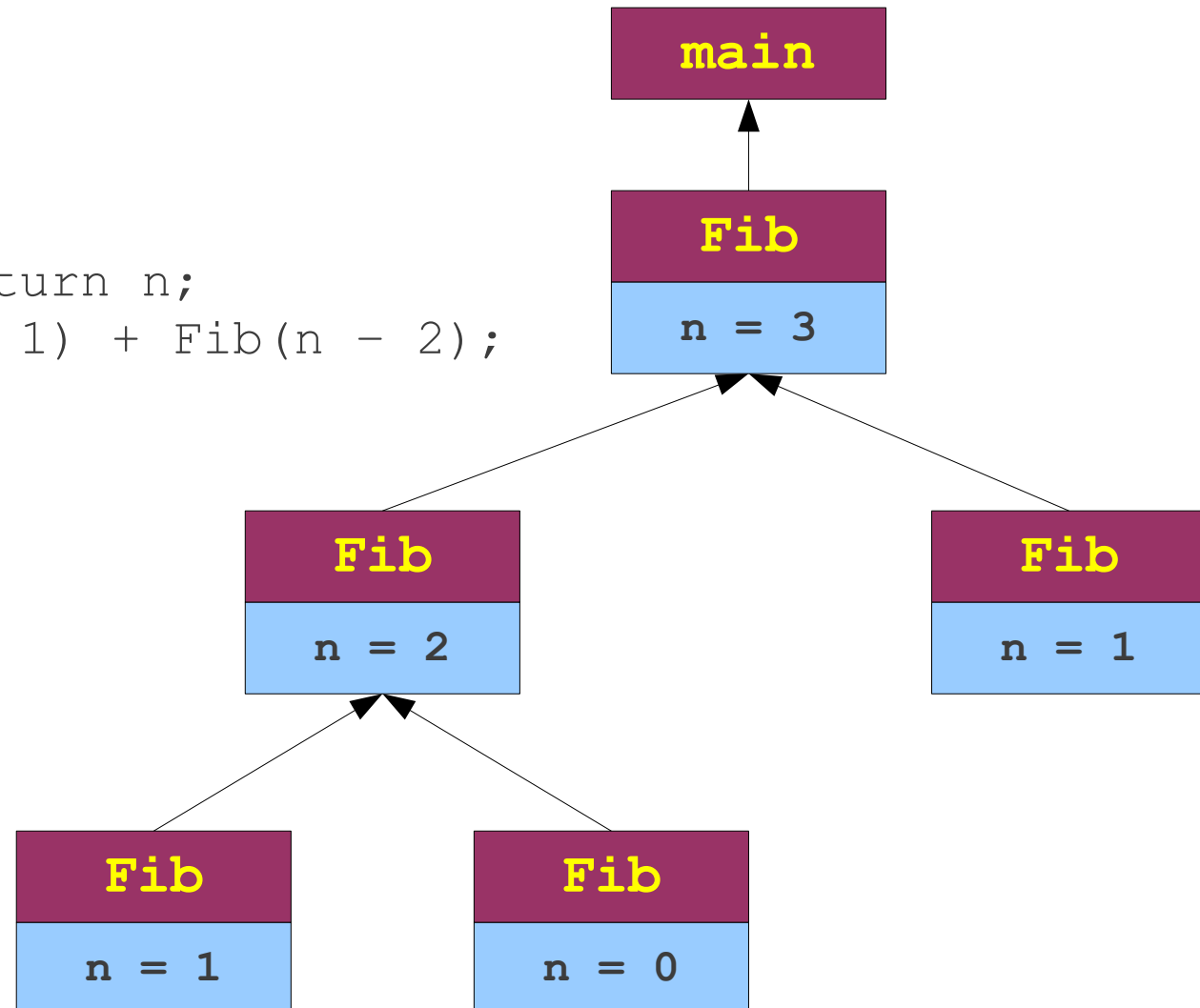
Activation Trees

- An **activation tree** is a tree structure representing all of the function calls made by a program on a particular execution.
 - Depends on the runtime behavior of a program; can't always be determined at compile-time.
 - (The static equivalent is the **call graph**).
- Each node in the tree is an activation record.
- Each activation record stores a **control link** to the activation record of the function that invoked it.

Activation Trees

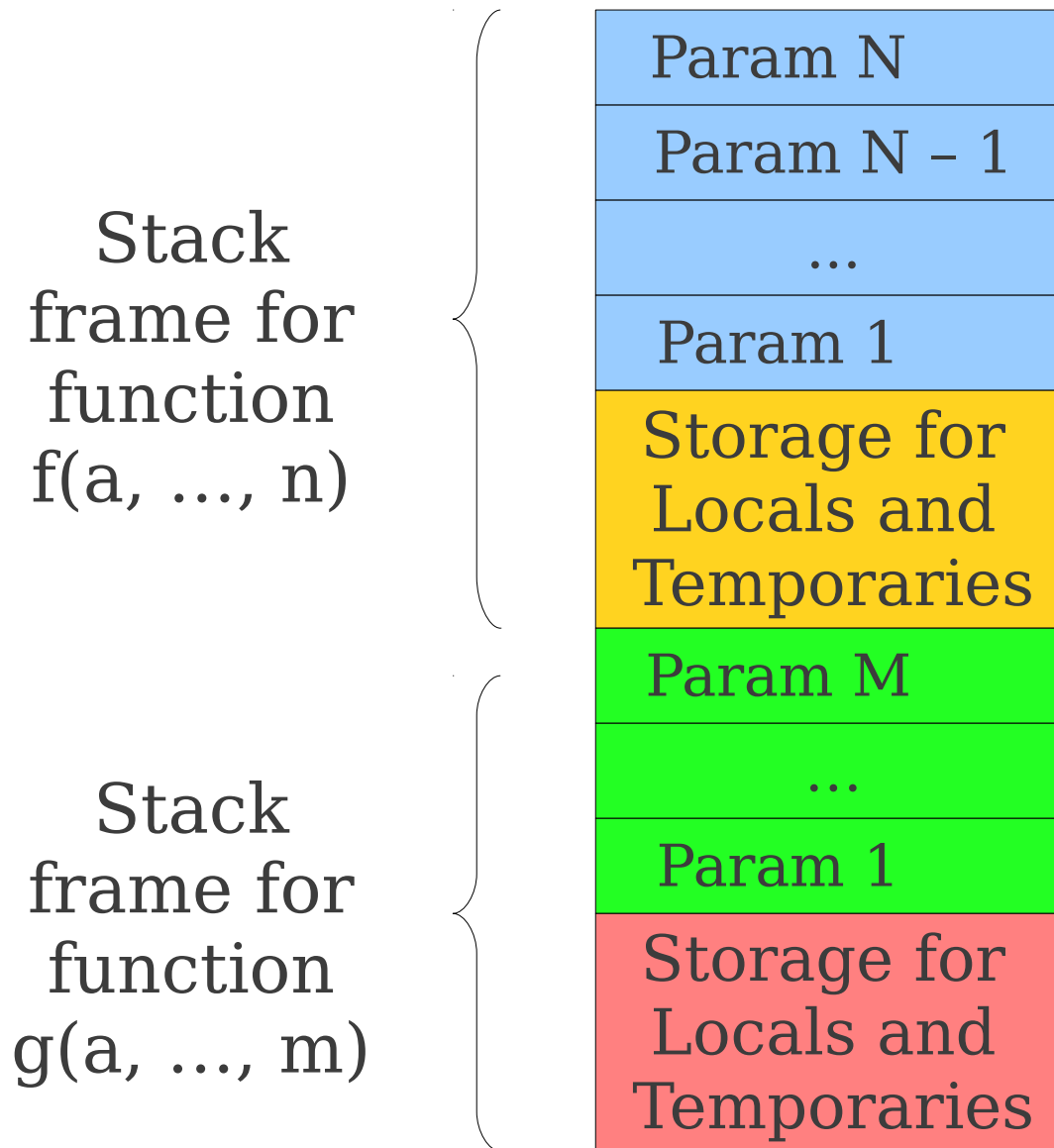
```
int main() {  
    Fib(3);  
}
```

```
int Fib(int n) {  
    if (n <= 1) return n;  
    return Fib(n - 1) + Fib(n - 2);  
}
```



The runtime stack is an **optimization**
of this spaghetti stack.

A Logical Decaf Stack Frame



Parameter Passing Approaches

- Two common approaches.
- Call-by-value
 - Parameters are *copies* of the values specified as arguments.
- Call-by-reference:
 - Parameters are *pointers* to values specified as parameters.

Implementing Objects

Objects are Hard

- It is difficult to build an *expressive* and *efficient* object-oriented language.
- Certain concepts are difficult to implement efficiently:
 - Dynamic dispatch (virtual functions)
 - Interfaces
 - Multiple Inheritance
 - Dynamic type checking (i.e. **instanceof**)
- Interfaces are so tricky to get right we won't ask you to implement them in PP4.

Encoding C-Style **structs**

- A **struct** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

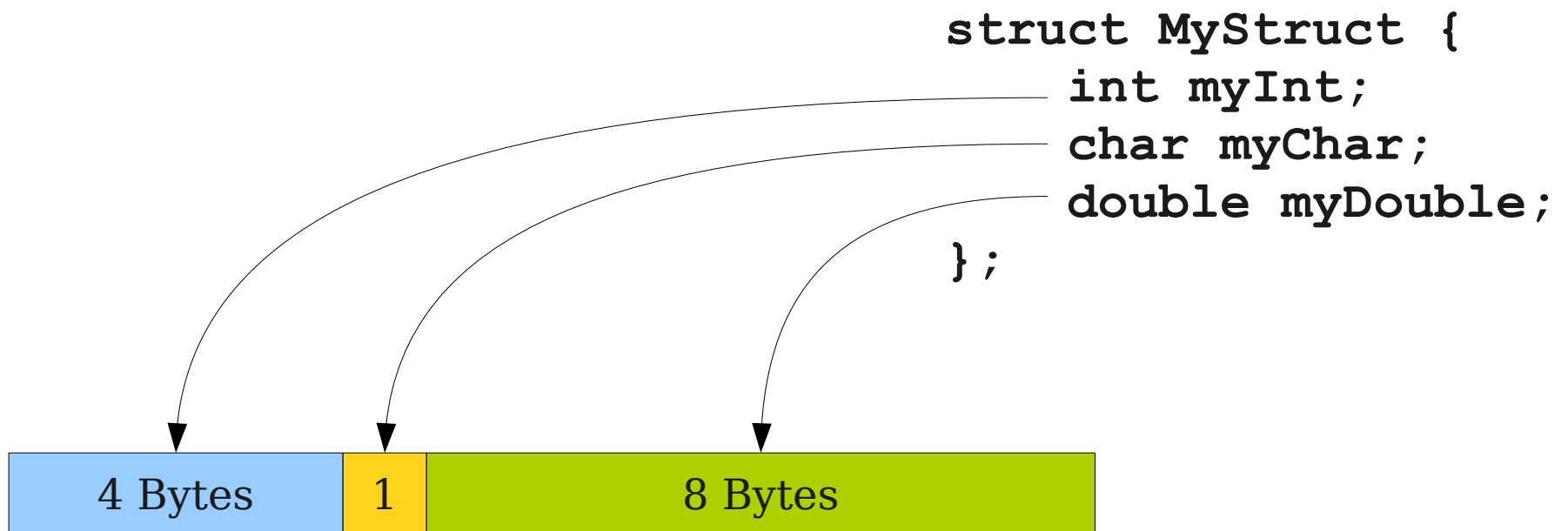
Encoding C-Style **structs**

- A **struct** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

```
struct MyStruct {  
    int myInt;  
    char myChar;  
    double myDouble;  
};
```

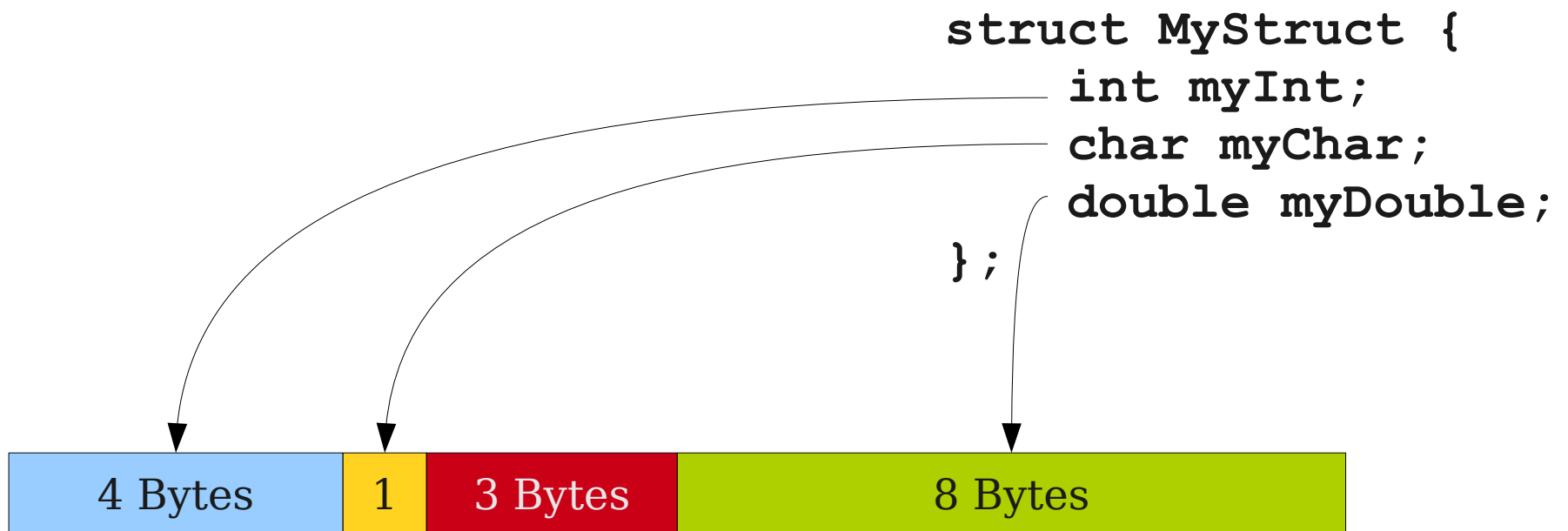
Encoding C-Style **structs**

- A **struct** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.



Encoding C-Style **structs**

- A **struct** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.



Accessing Fields

- Once an object is laid out in memory, it's just a series of bytes.
- How do we know where to look to find a particular field?



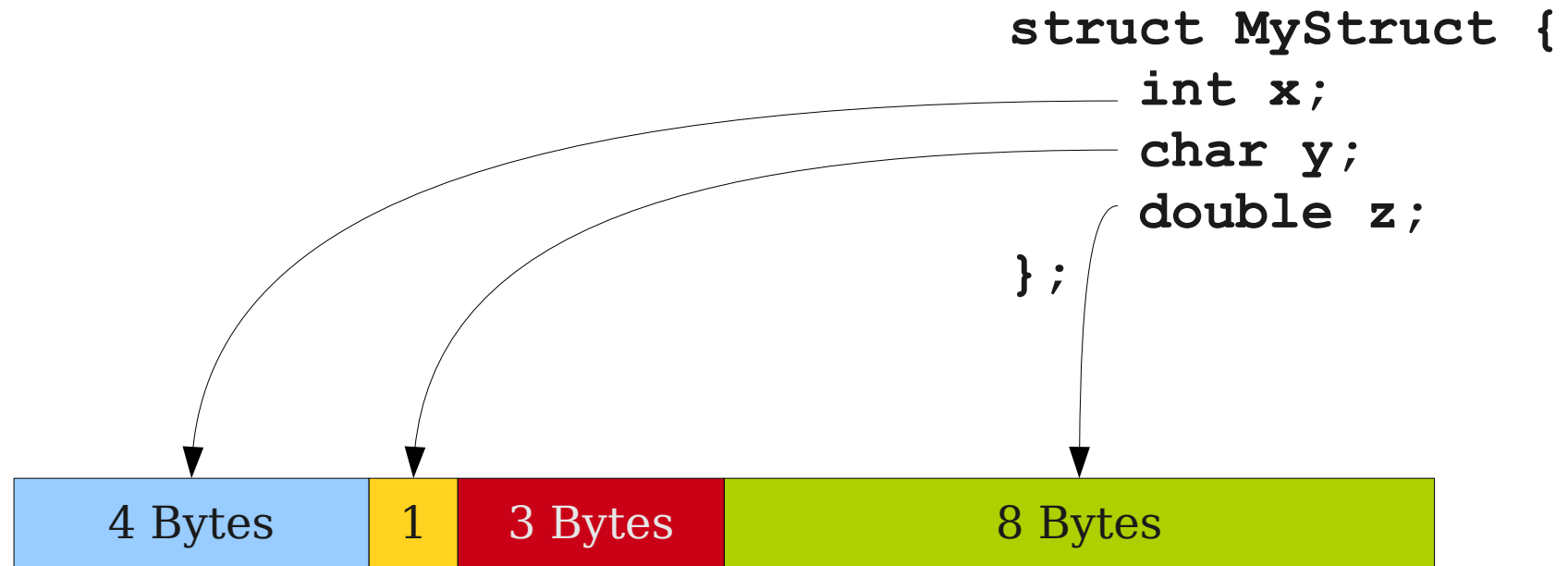
Accessing Fields

- Once an object is laid out in memory, it's just a series of bytes.
- How do we know where to look to find a particular field?

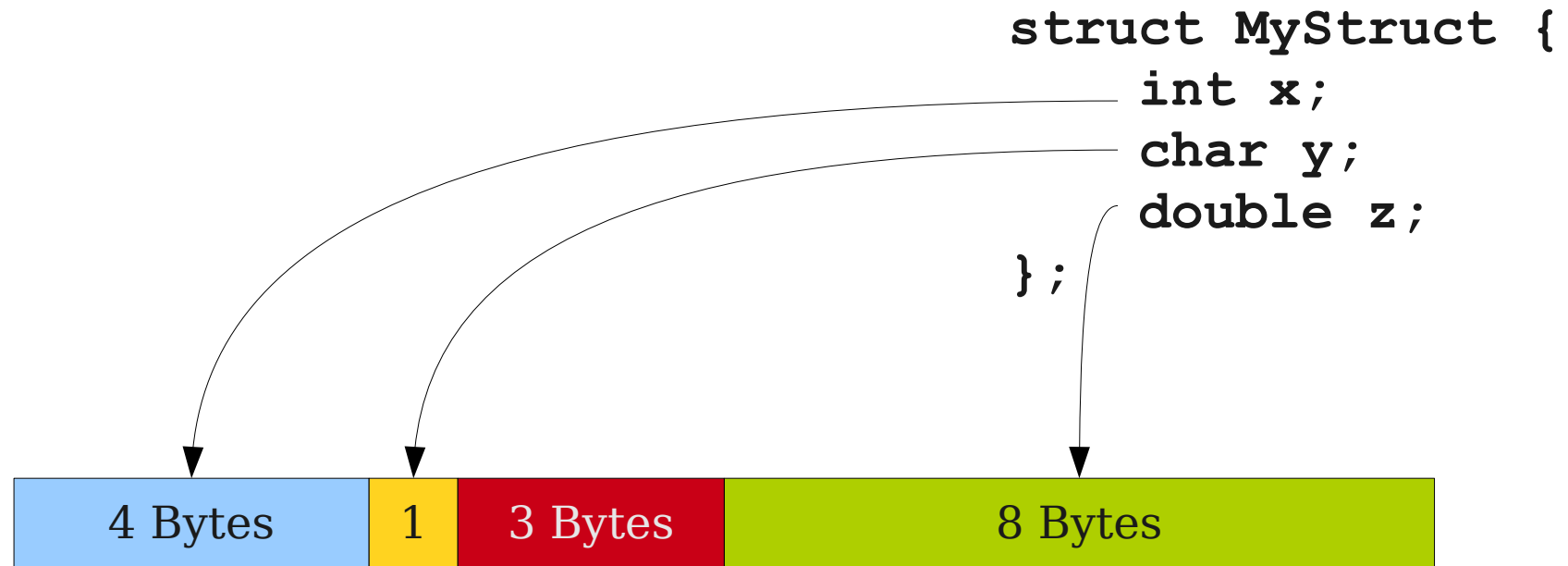


- Idea: Keep an internal table inside the compiler containing the offsets of each field.
- To look up a field, start at the base address of the object and advance forward by the appropriate offset.

Field Lookup

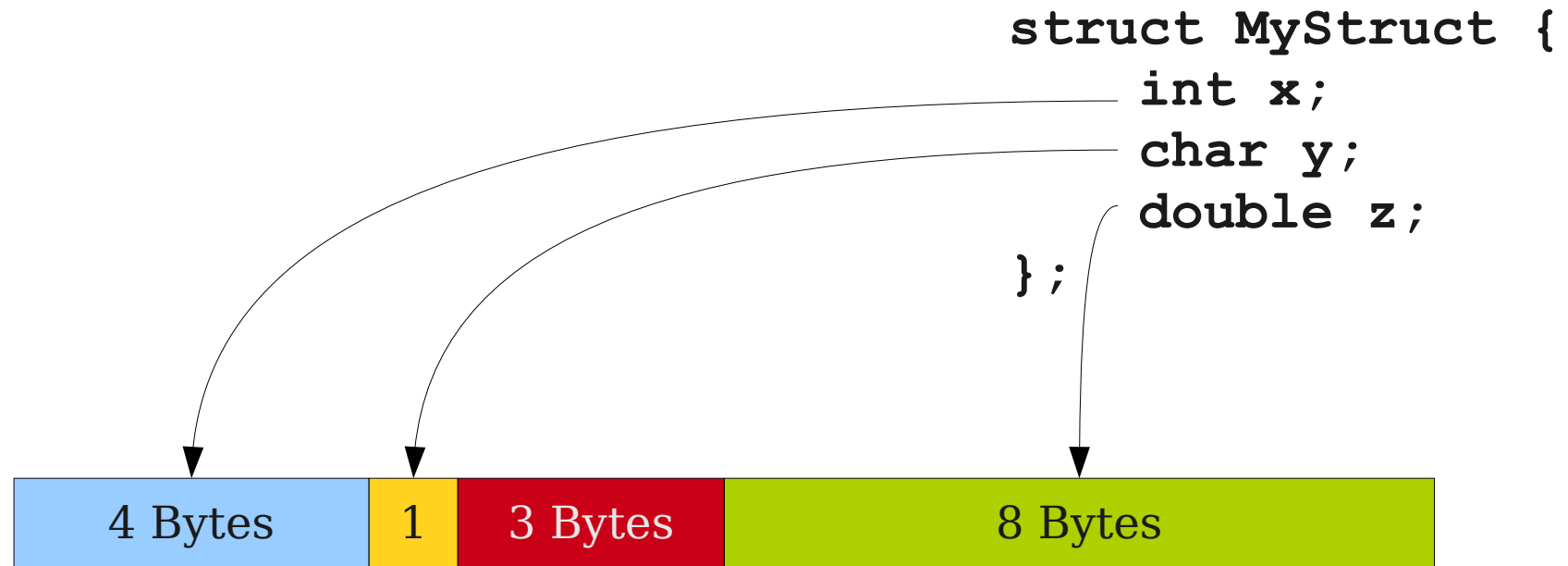


Field Lookup



```
MyStruct* ms = new MyStruct;  
ms->x = 137;  
ms->y = 'A';  
ms->z = 2.71
```

Field Lookup



```
MyStruct* ms = new MyStruct;  
ms->x = 137;    store 137    0 bytes after ms  
ms->y = 'A';    store 'A'    4 bytes after ms  
ms->z = 2.71    store 2.71  8 bytes after ms
```

OOP without Methods

- Consider the following Decaf code:

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

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

Memory Layouts with Inheritance

Memory Layouts with Inheritance

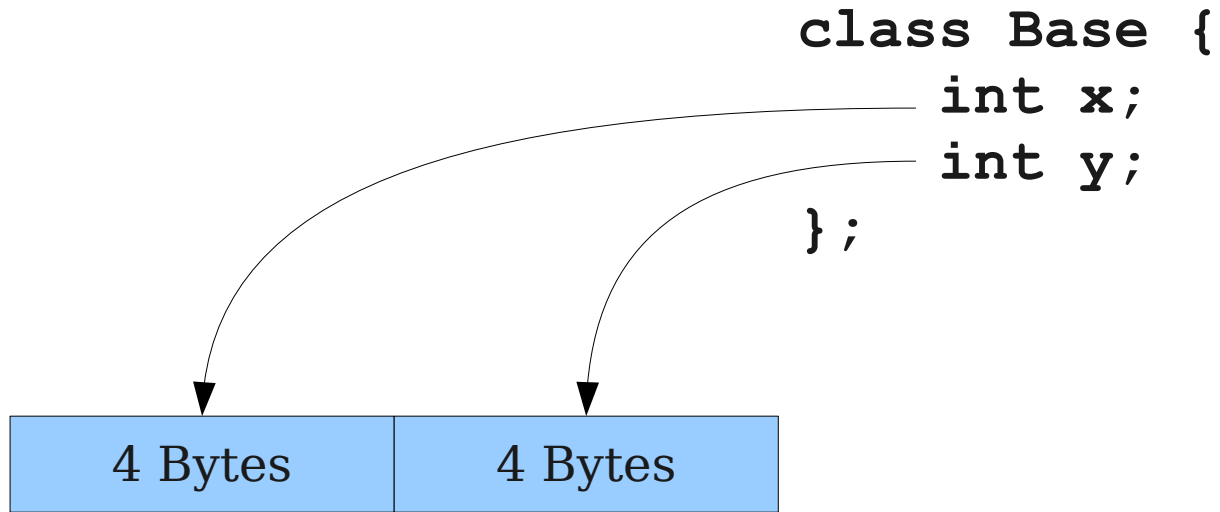
```
class Base {  
    int x;  
    int y;  
};
```

Memory Layouts with Inheritance

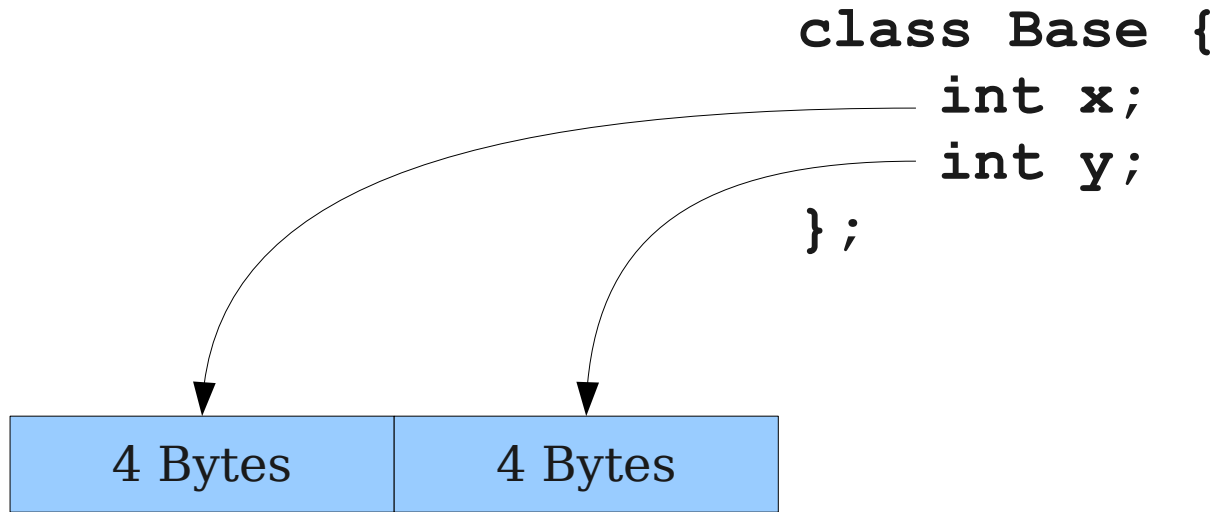
```
class Base {  
    int x;  
    int y;  
};
```

4 Bytes	4 Bytes
---------	---------

Memory Layouts with Inheritance

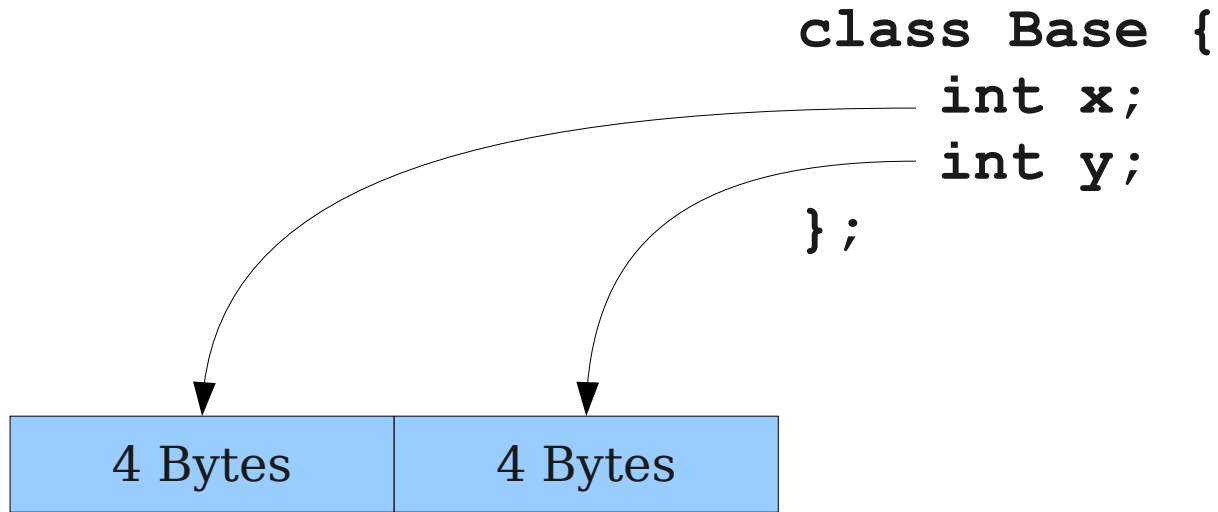


Memory Layouts with Inheritance



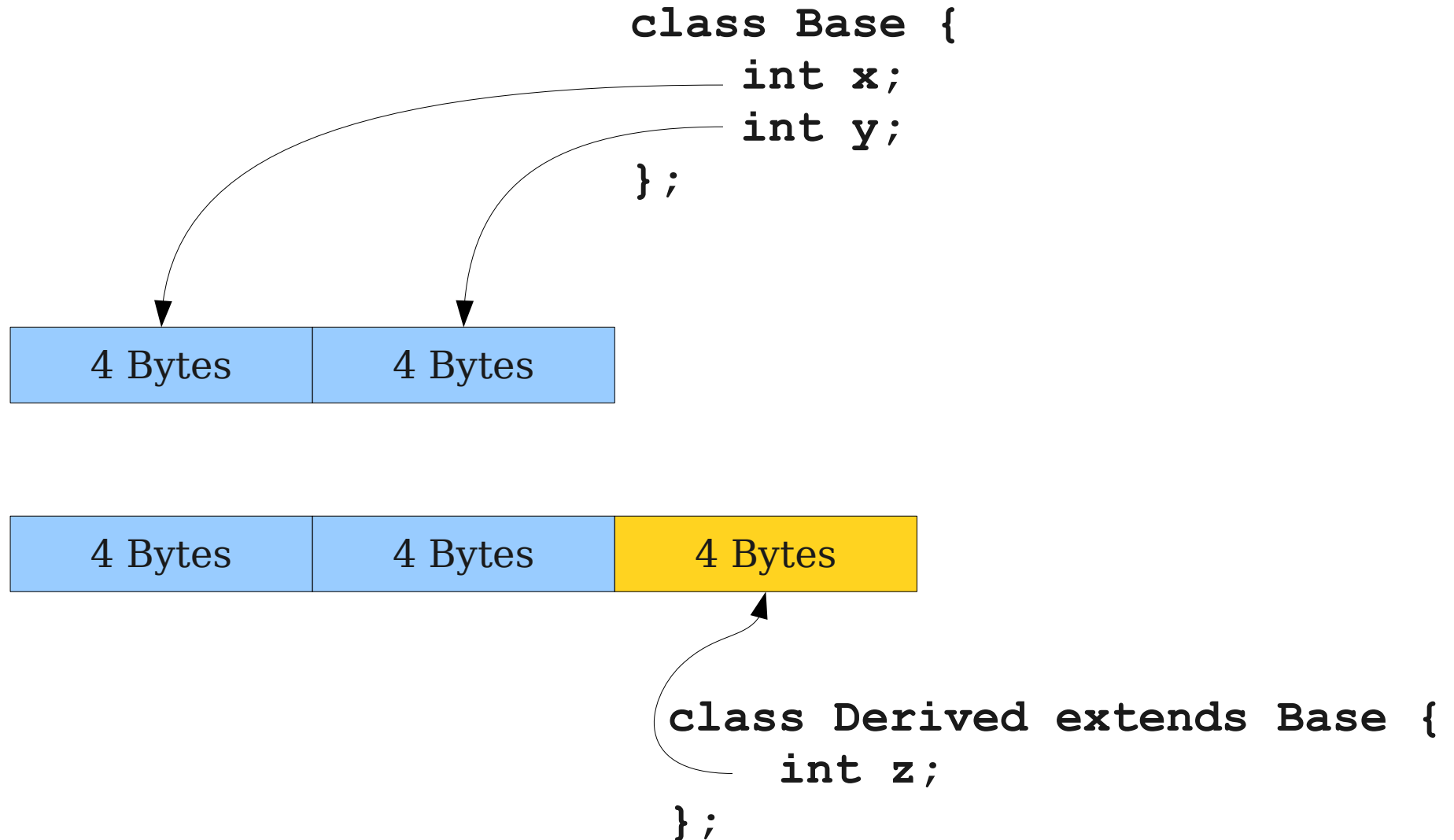
```
class Derived extends Base {  
    int z;  
};
```

Memory Layouts with Inheritance



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

Memory Layouts with Inheritance

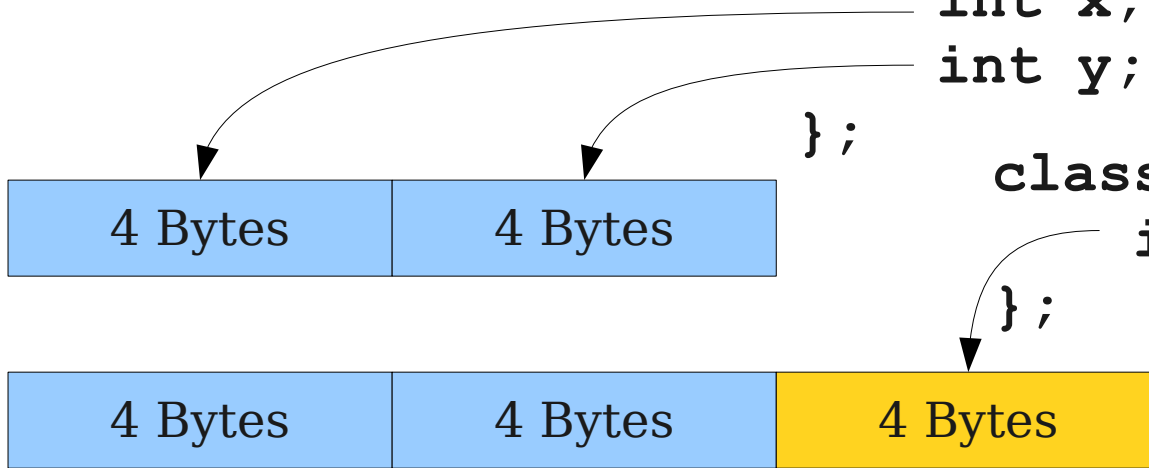


Field Lookup With Inheritance

Field Lookup With Inheritance

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

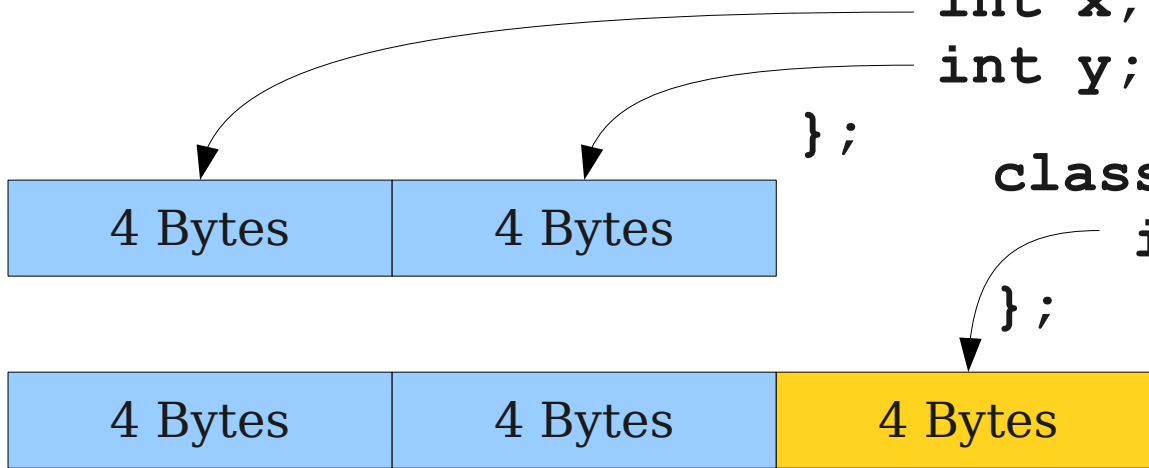
```
class Derived extends Base {  
    int z;  
};
```



Field Lookup With Inheritance

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

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

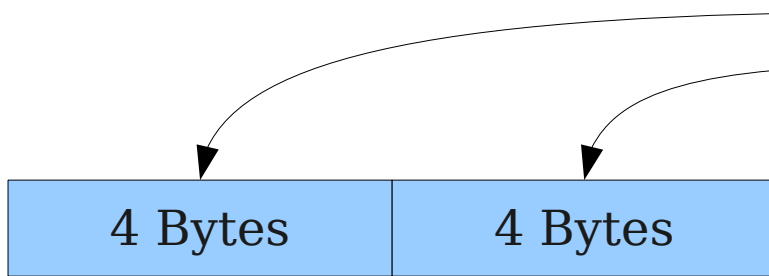


```
Base ms = new Base;  
ms.x = 137;  
ms.y = 42;
```

Field Lookup With Inheritance

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

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



```
Base ms = new Base;
```

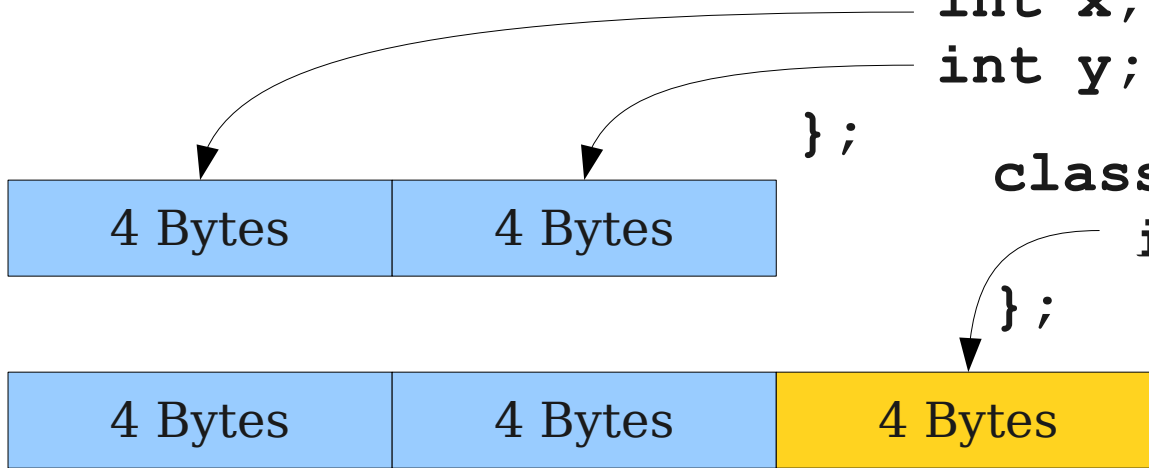
```
ms.x = 137;      store 137 0 bytes after ms
```

```
ms.y = 42;       store 42  4 bytes after ms
```

Field Lookup With Inheritance

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

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

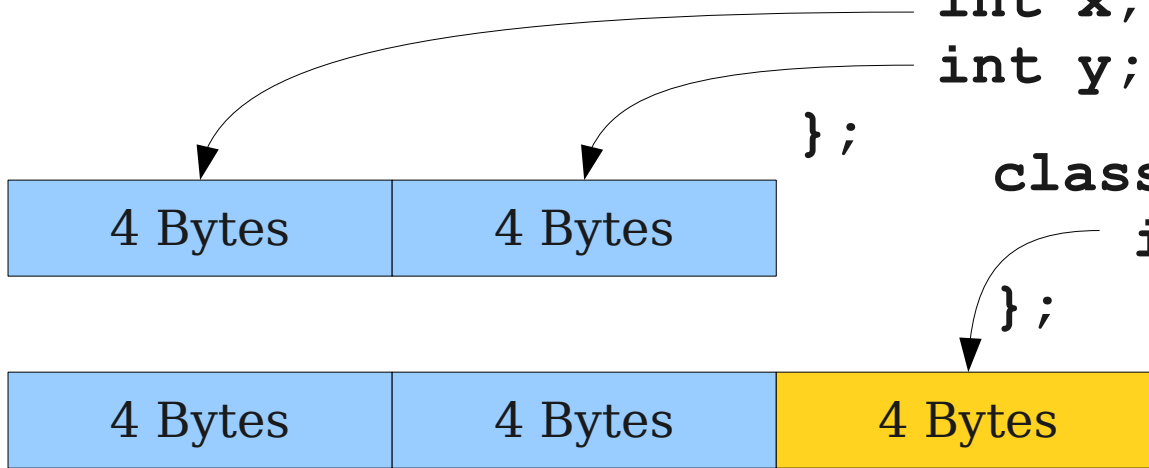


```
Base ms = new Derived;  
ms.x = 137;  
ms.y = 42;
```

Field Lookup With Inheritance

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

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

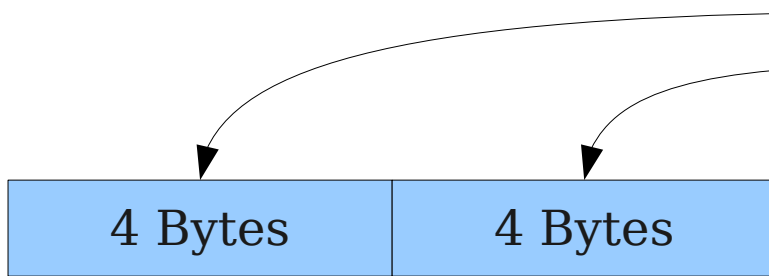


```
Base ms = new Derived;  
ms.x = 137;  
ms.y = 42;
```

Field Lookup With Inheritance

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

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

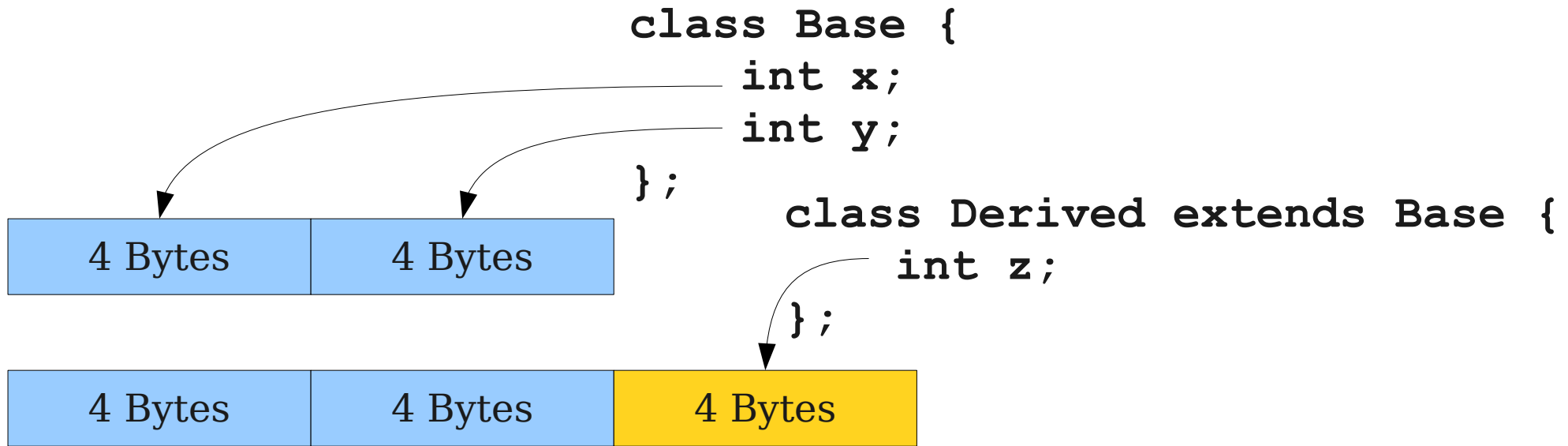


```
Base ms = new Derived;
```

```
ms.x = 137;      store 137 0 bytes after ms
```

```
ms.y = 42;       store 42 4 bytes after ms
```

Field 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
```

Single Inheritance in Decaf

- The memory layout for a class D that extends B is given by the memory layout for B followed by the memory layout for the members of D.
 - Actually a bit more complex; we'll see why later.
- 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 Member Functions?

- Member functions 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)?

`this` is Tricky

- 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 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;  
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);
```

this Rules

- When generating code to call a member function, remember to pass some object as the **this** parameter representing the receiver object.
- Inside of a 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.

Implementing Dynamic Dispatch

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

- This previous idea has several serious problems.
- What are they?
- **It's 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?

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.

Layout of **Base**



Layout of **Derived**



- Can we do something similar with functions?

Virtual Function Tables

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

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

Virtual Function Tables

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

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

Code for
Base.sayHi

Code for
Derived.sayHi

Virtual Function Tables

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

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

Base.x

Code for
Base.sayHi

Code for
Derived.sayHi

Virtual Function Tables

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

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

Base.x

Base.x

Derived.y

Code for
Base.sayHi

Code for
Derived.sayHi

Virtual Function Tables

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

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

sayHi	Base.x
--------------	--------

sayHi	Base.x	Derived.y
--------------	--------	-----------

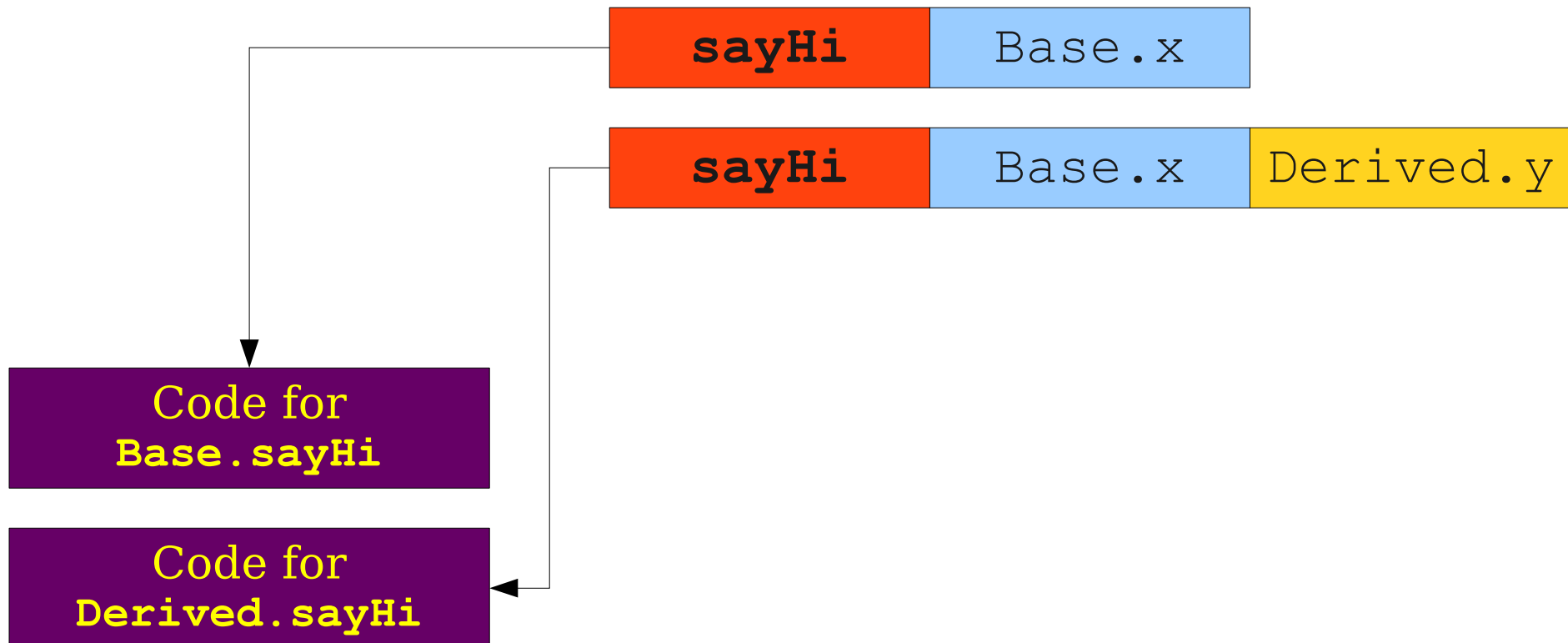
Code for
Base.sayHi

Code for
Derived.sayHi

Virtual Function Tables

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

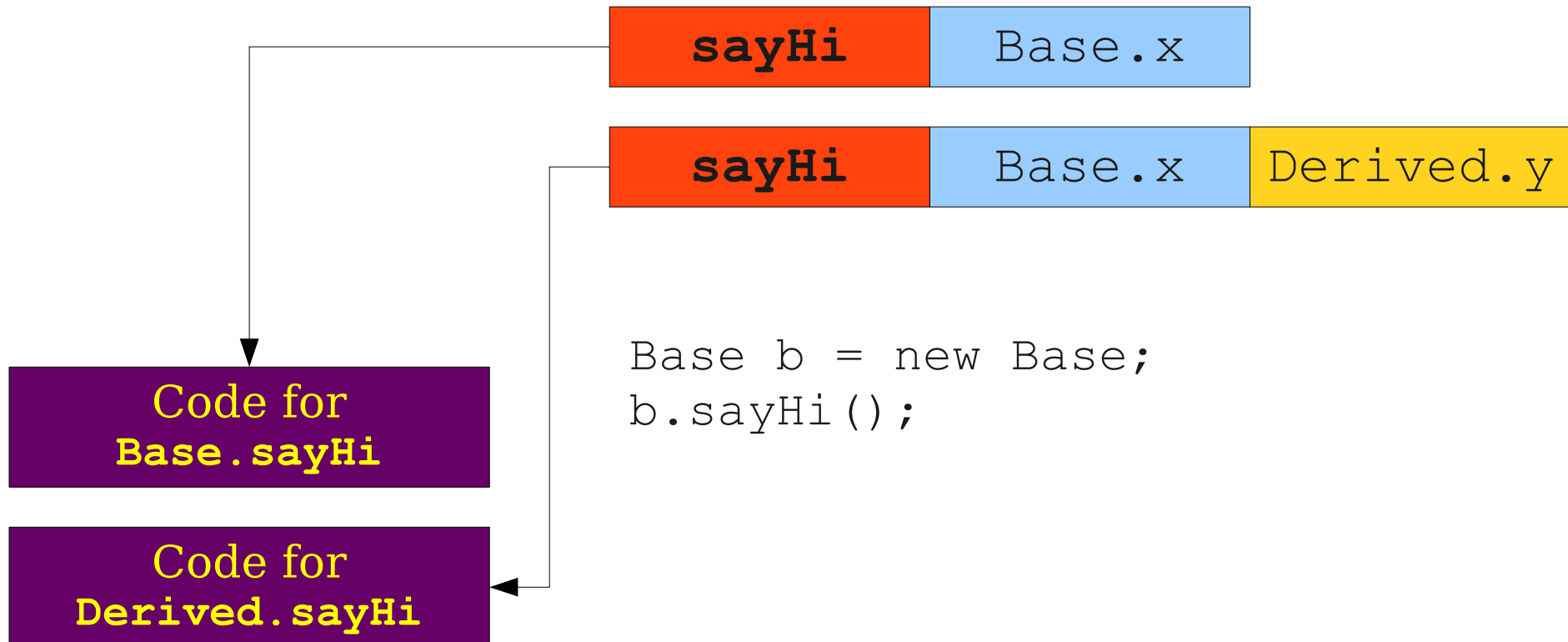
```
class Derived extends Base {  
    int y;  
    void sayHi() {  
        Print("Derived");  
    }  
}
```



Virtual Function Tables

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

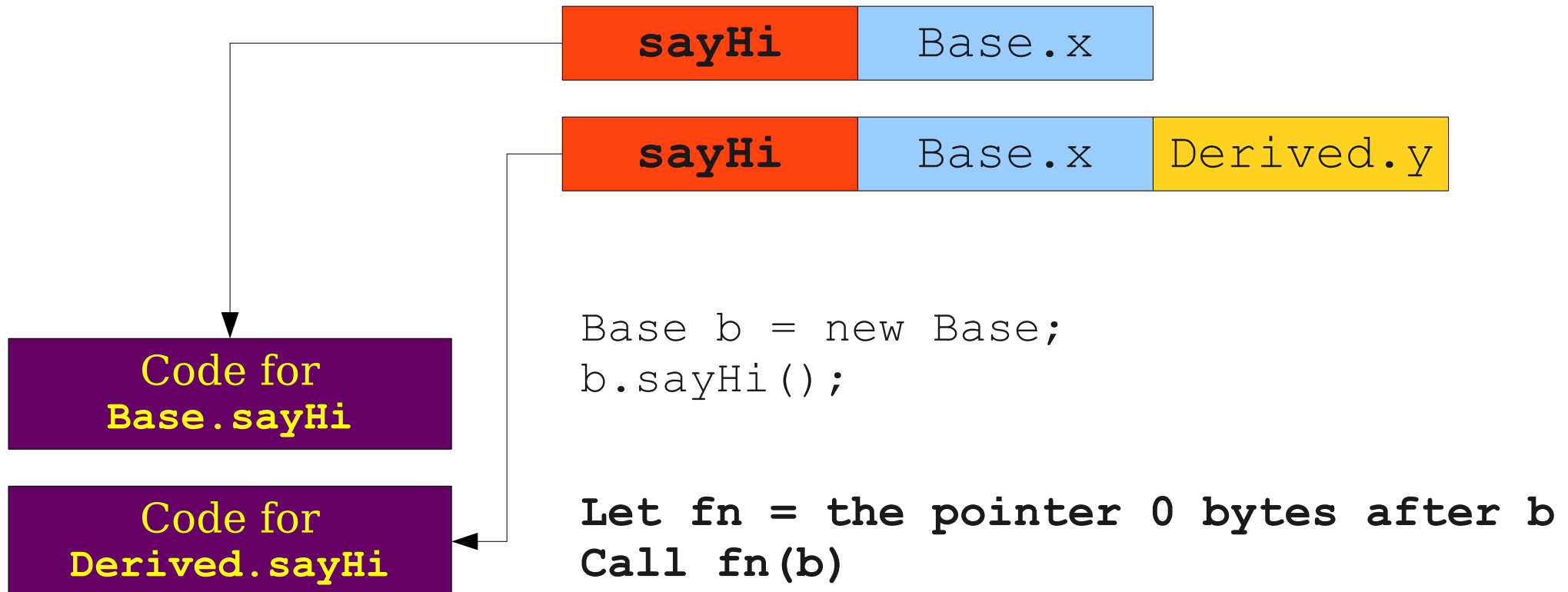
```
class Derived extends Base {  
    int y;  
    void sayHi() {  
        Print("Derived");  
    }  
}
```



Virtual Function Tables

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

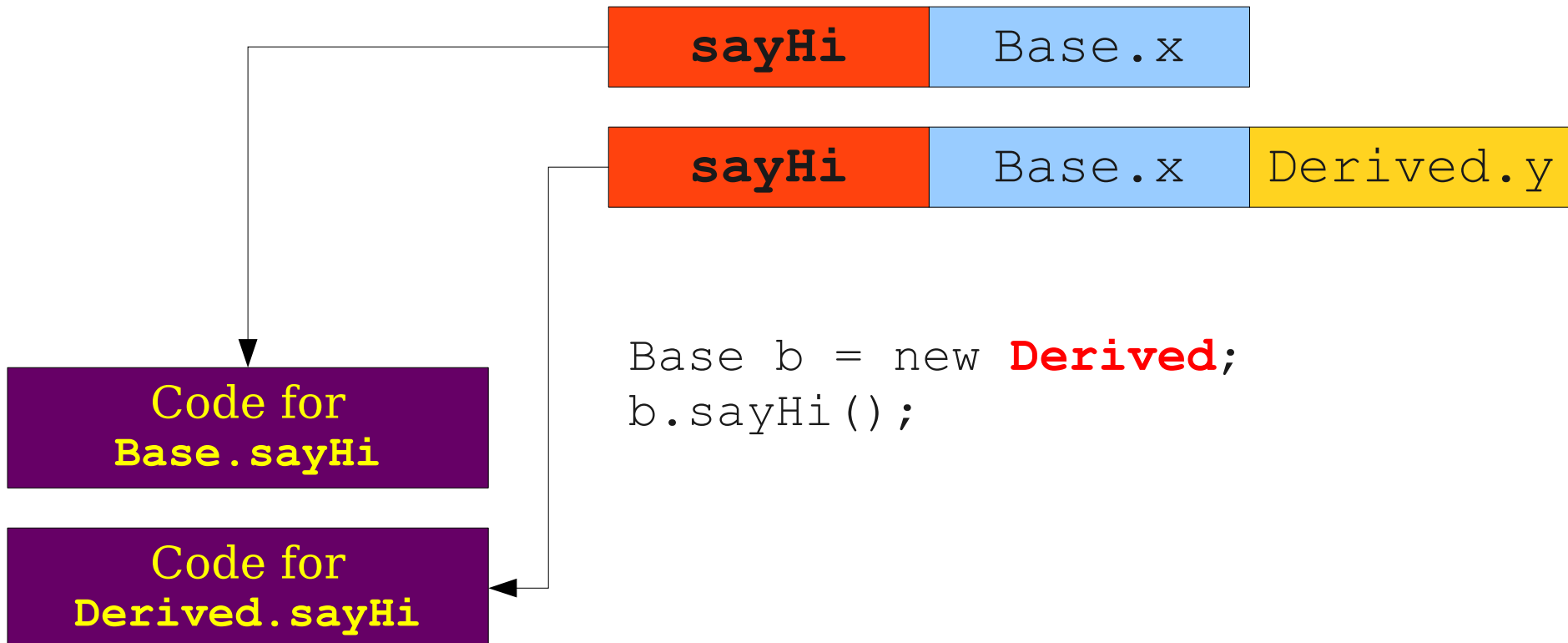
```
class Derived extends Base {  
    int y;  
    void sayHi() {  
        Print("Derived");  
    }  
}
```



Virtual Function Tables

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

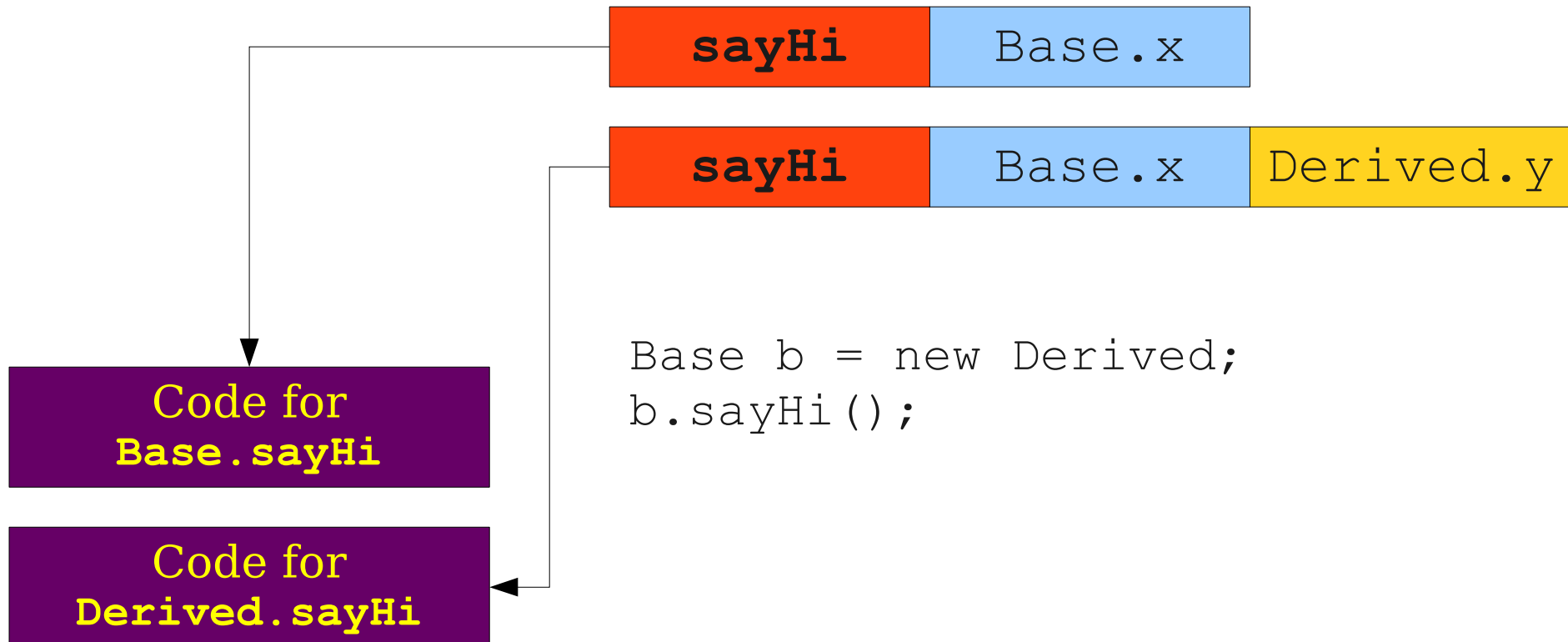
```
class Derived extends Base {  
    int y;  
    void sayHi() {  
        Print("Derived");  
    }  
}
```



Virtual Function Tables

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

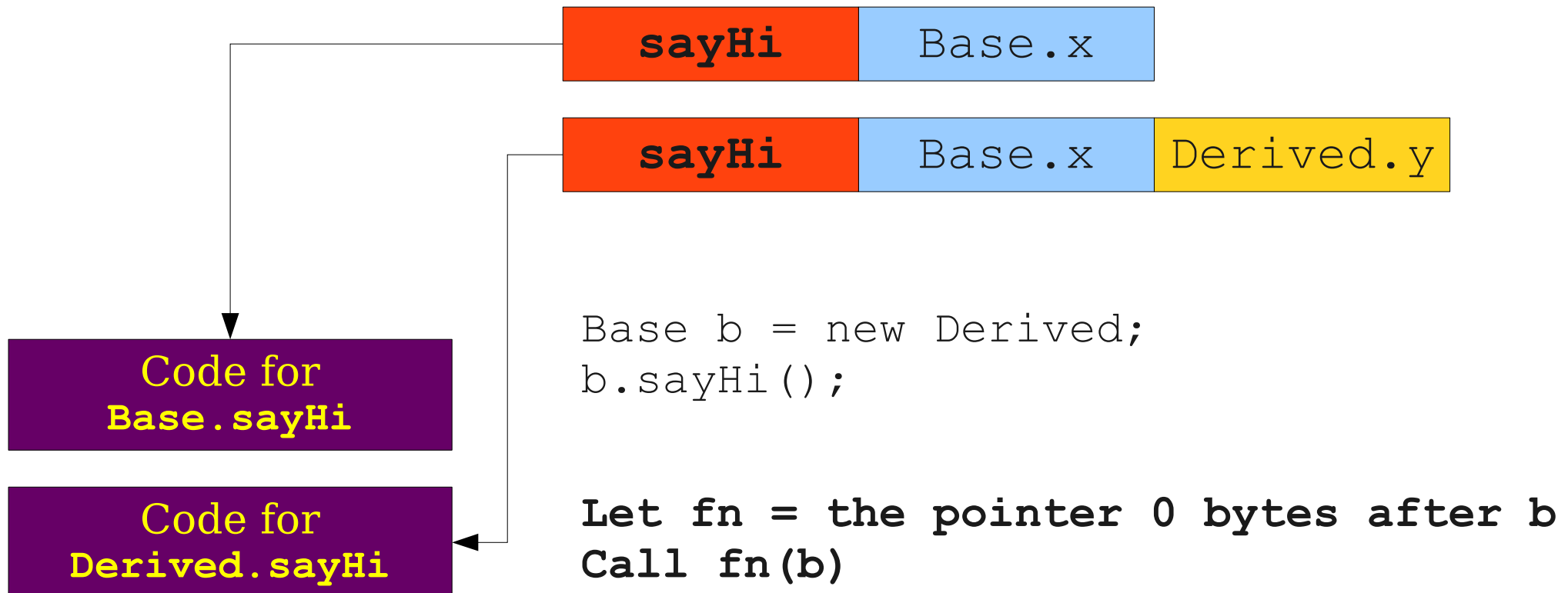
```
class Derived extends Base {  
    int y;  
    void sayHi() {  
        Print("Derived");  
    }  
}
```



Virtual Function Tables

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

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



More Virtual Function 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 Virtual Function Tables

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

Code for
Base.sayHi

Code for
Base.clone

Code for
Derived.clone

```
class Derived extends Base {  
    int y;  
  
    Derived clone() {  
        return new Derived;  
    }  
}
```

More Virtual Function Tables

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

Code for
Base.sayHi

Code for
Base.clone

Code for
Derived.clone

```
class Derived extends Base {  
    int y;  
  
    Derived clone() {  
        return new Derived;  
    }  
}
```

sayHi

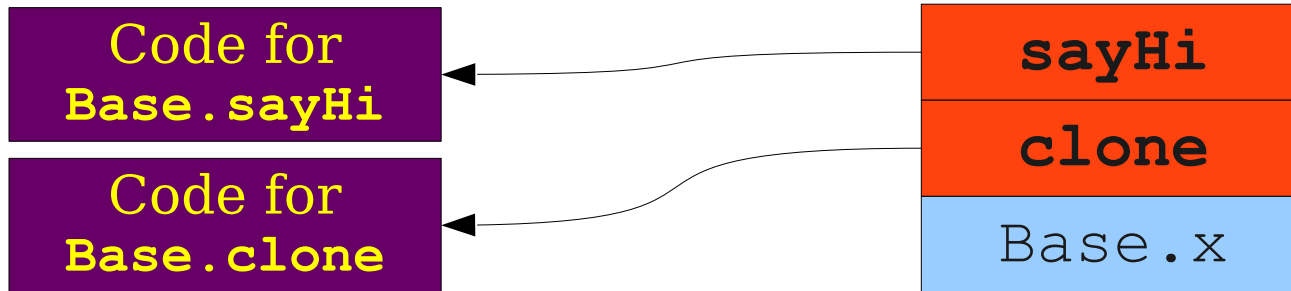
clone

Base.x

More Virtual Function 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;  
    }  
}
```

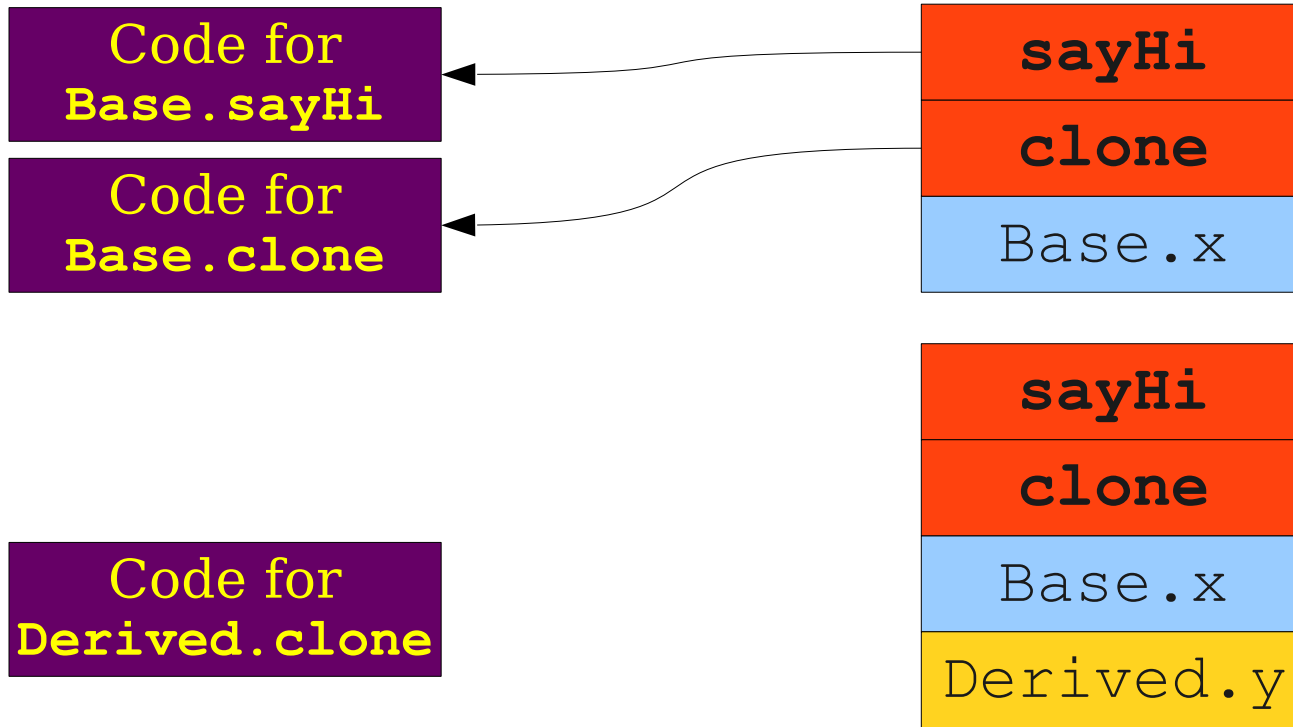


Code for
Derived.clone

More Virtual Function Tables

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

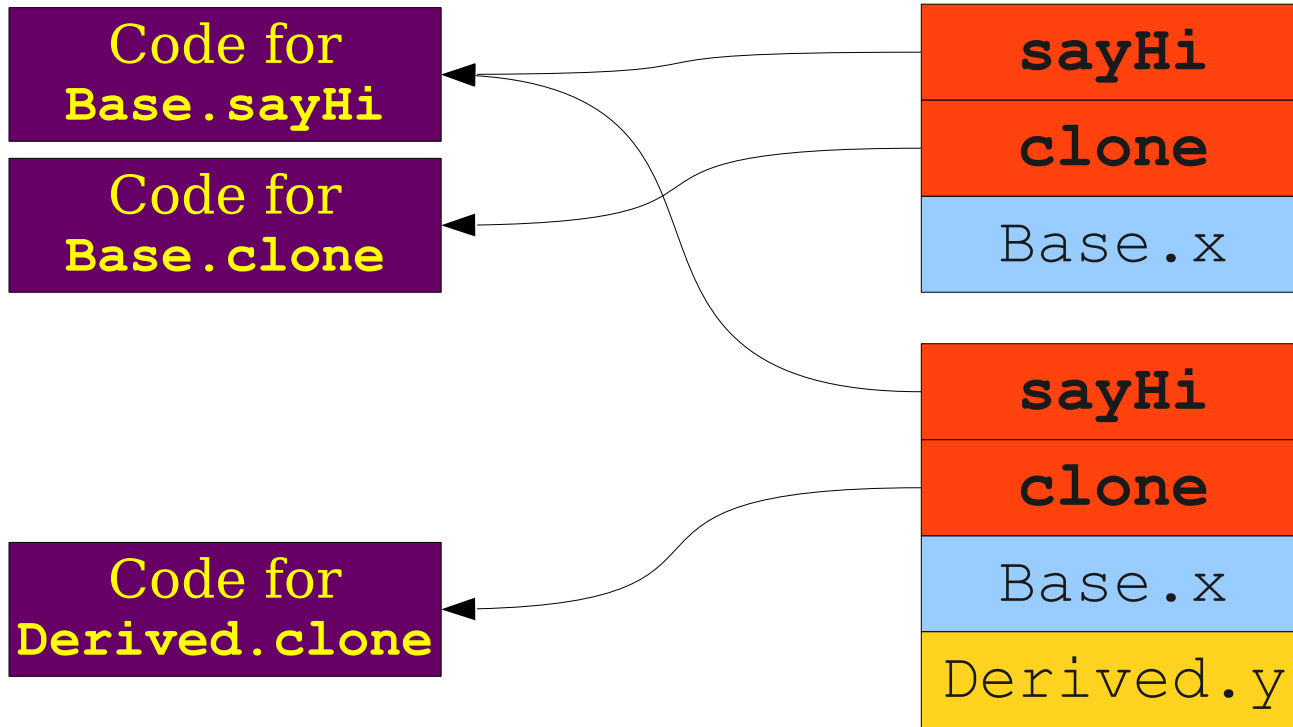
```
class Derived extends Base {  
    int y;  
  
    Derived clone() {  
        return new Derived;  
    }  
}
```



More Virtual Function 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;  
    }  
}
```



Virtual Function Tables

- A **virtual function table** (or **vtable**) is an array of pointers to the member function implementations for a particular class.
- To invoke a member function:
 - Determine (statically) its index in the vtable.
 - Follow the pointer at that index in the object's vtable to the code for the function.
 - Invoke that function.

Analyzing our Approach

- Advantages:
 - Time to determine function to call is $O(1)$.
 - (and a good $O(1)$ too!)
- What are the disadvantages?

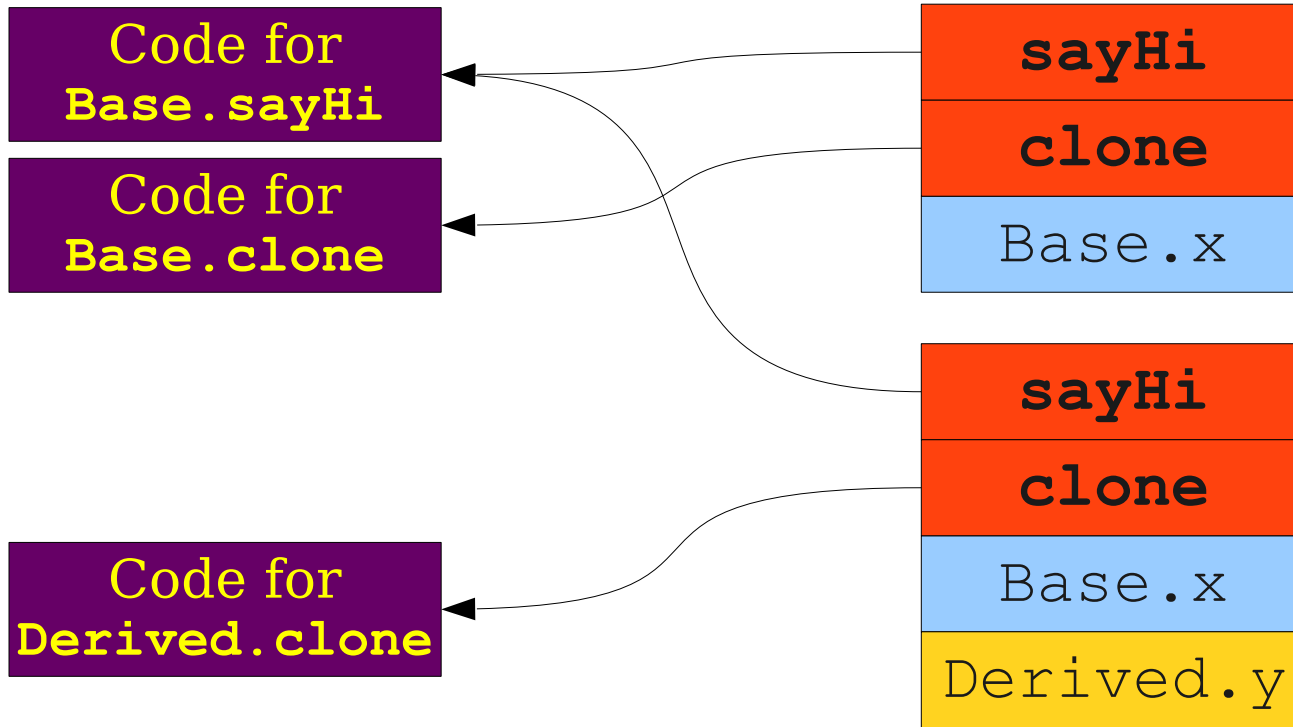
Analyzing our Approach

- Advantages:
 - Time to determine function to call is $O(1)$.
 - (and a good $O(1)$ too!)
- What are the disadvantages?
- **Object sizes are larger.**
 - Each object needs to have space for $O(M)$ pointers.
- **Object creation is slower.**
 - Each new object needs to have $O(M)$ pointers set, where M is the number of member functions.

A Common Optimization

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

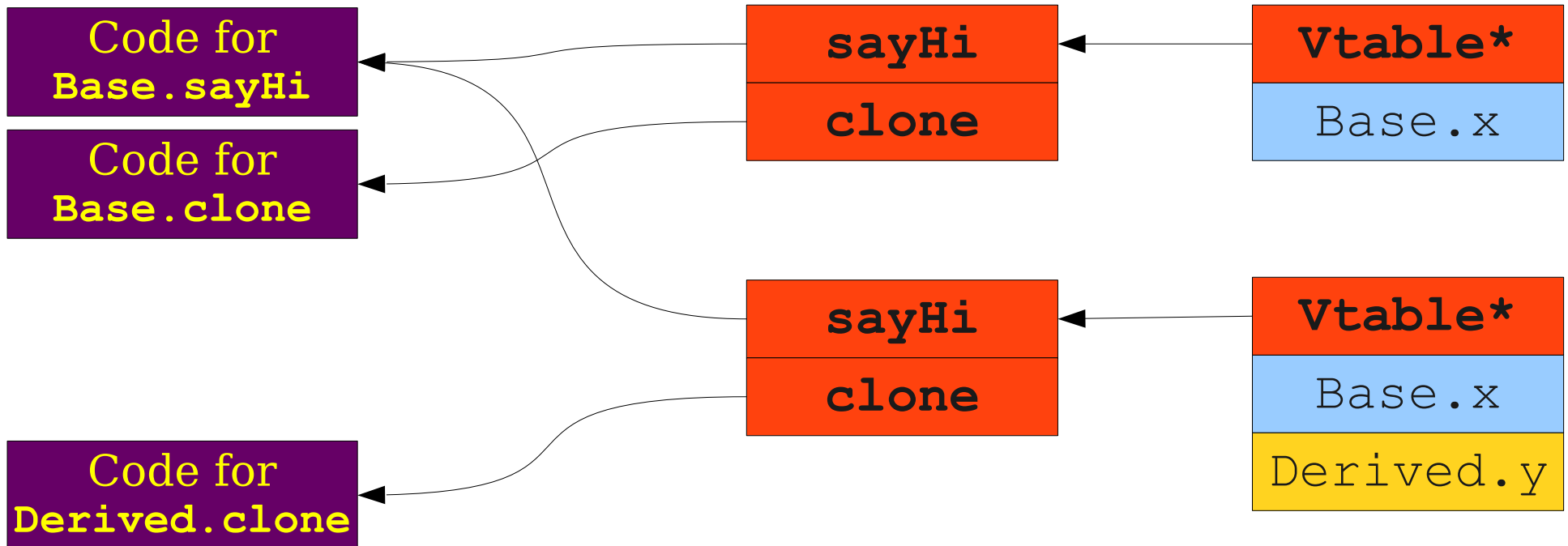
```
class Derived extends Base {  
    int y;  
  
    Derived clone() {  
        return new Derived;  
    }  
}
```



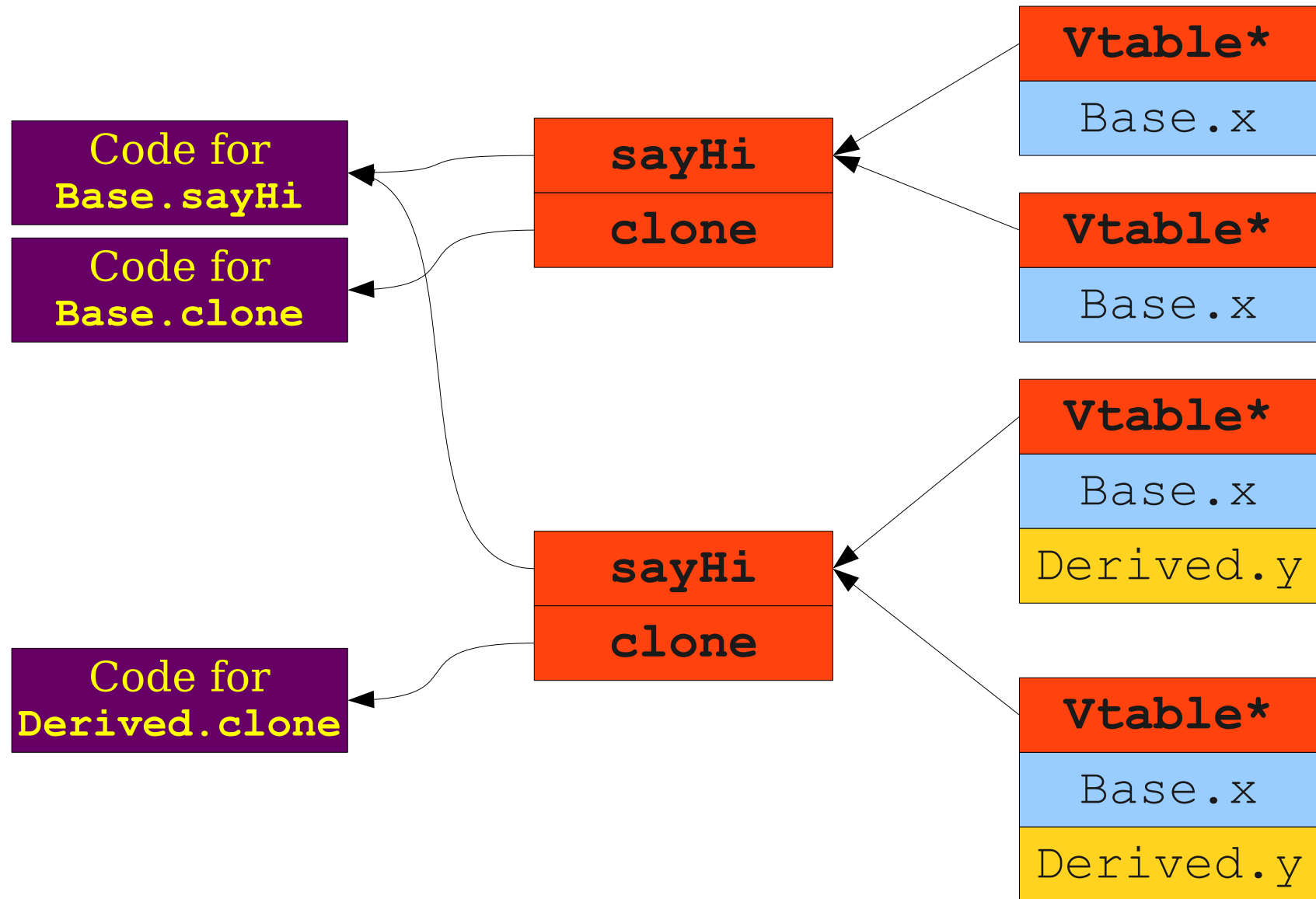
A Common Optimization

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

```
class Derived extends Base {  
    int y;  
  
    Derived clone() {  
        return new Derived;  
    }  
}
```



Objects in Memory



Dynamic Dispatch in $O(1)$

- Create a single instance of the vtable for each class.
- Have each object store a pointer to the vtable.
- Can follow the pointer to the table in $O(1)$.
- Can index into the table in $O(1)$.
- Can set the vtable pointer of a 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.**

Vtable Requirements

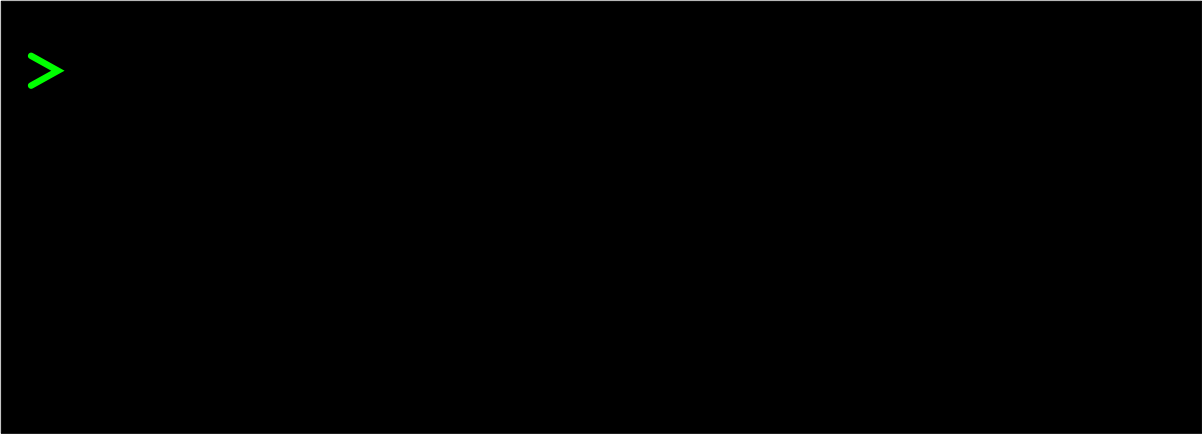
- We've made implicit assumptions about our language that allow vtables to work correctly.
- What are they?
- **Method calls known statically.**
 - We can determine at compile-time which methods are intended at each call (even if we're not sure which method is ultimately invoked).
- **Single inheritance.**
 - Don't need to worry about building a single vtable for multiple different classes.

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}  
  
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

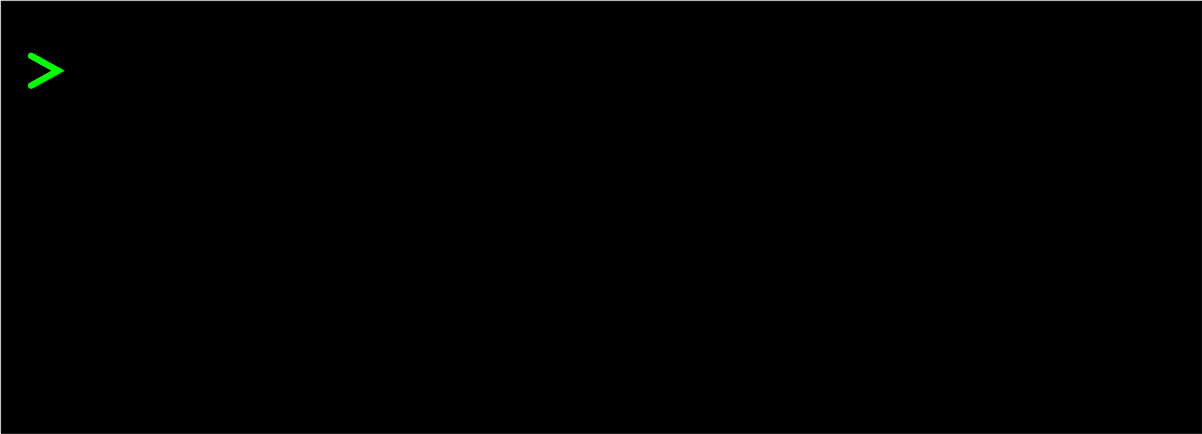
Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi! I'm Base.";  
    }  
}  
  
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi! I'm Derived.";  
    }  
}
```



Inheritance in PHP

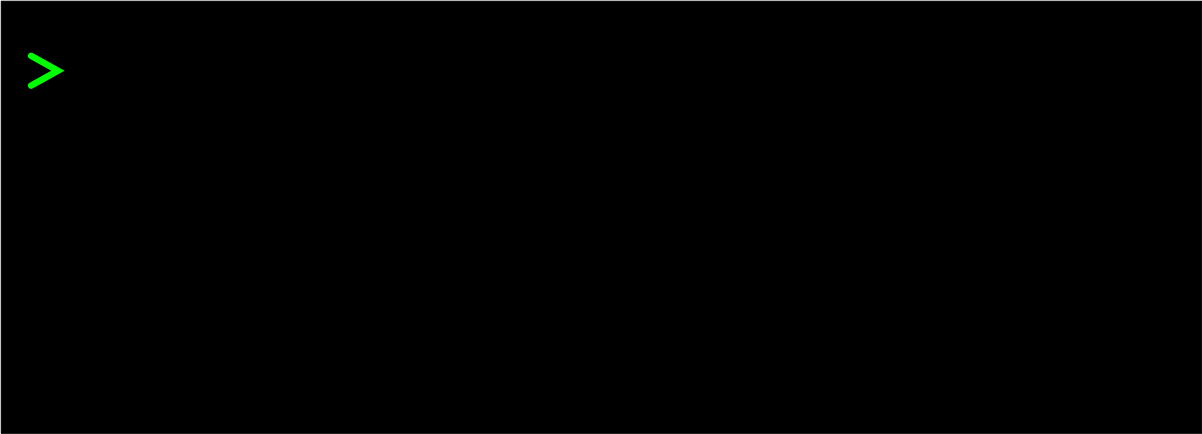
```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}  
  
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}  
  
$b = new Base();  
$b->sayHello();
```



Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}  
  
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```



Inheritance in PHP

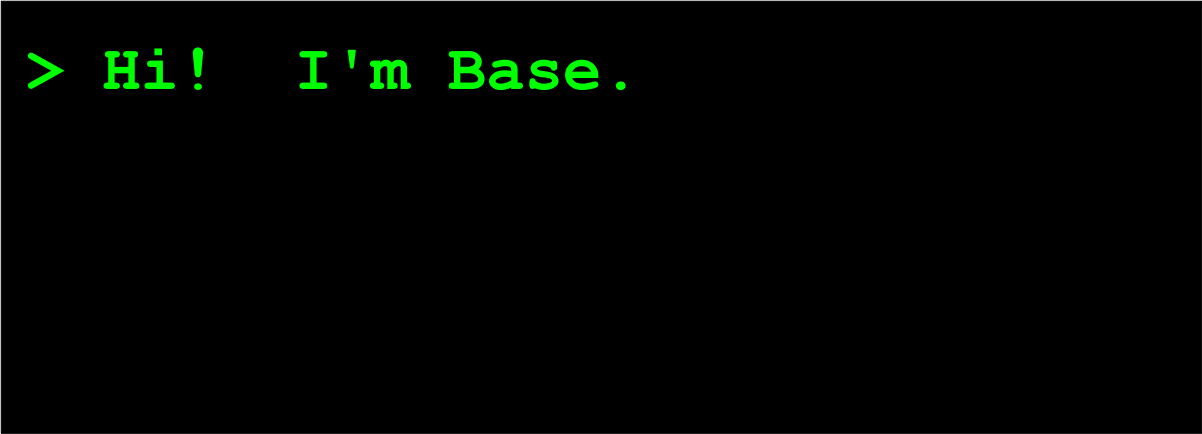
```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}  
  
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
> Hi!  I'm Base.
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}  
  
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}  
  
$b = new Base();  
$b->sayHello();
```



```
> Hi!  I'm Base.
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

A black rectangular box representing a terminal window. Inside, the text "> Hi! I'm Base." is displayed in a bright green monospace font, indicating the output of the first PHP execution.

> Hi! I'm Base.

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```



```
> Hi!  I'm Base.
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi! I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi! I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
> Hi! I'm Base.  
Hi! I'm Derived.
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi! I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi! I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
> Hi! I'm Base.  
Hi! I'm Derived.
```

Inheritance in PHP

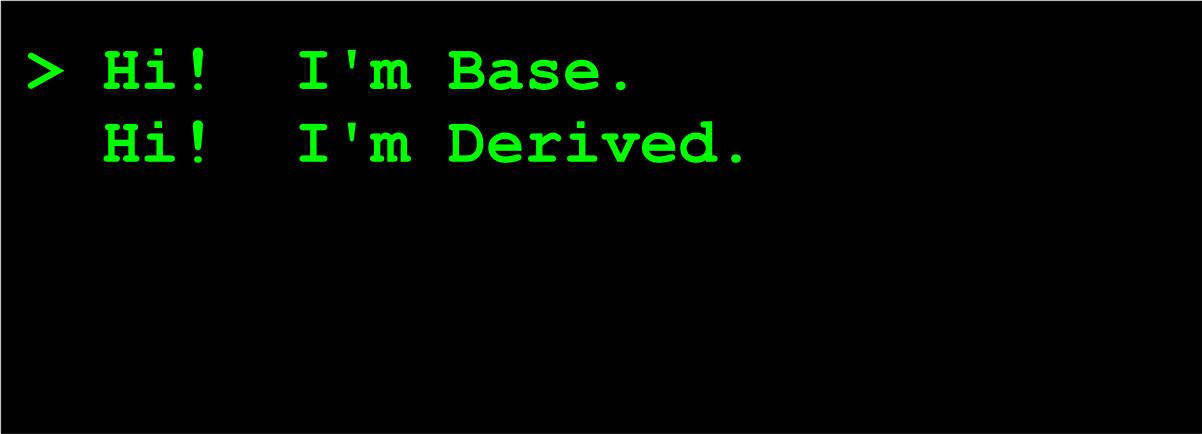
```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```



```
> Hi!  I'm Base.  
Hi!  I'm Derived.
```

Inheritance in PHP

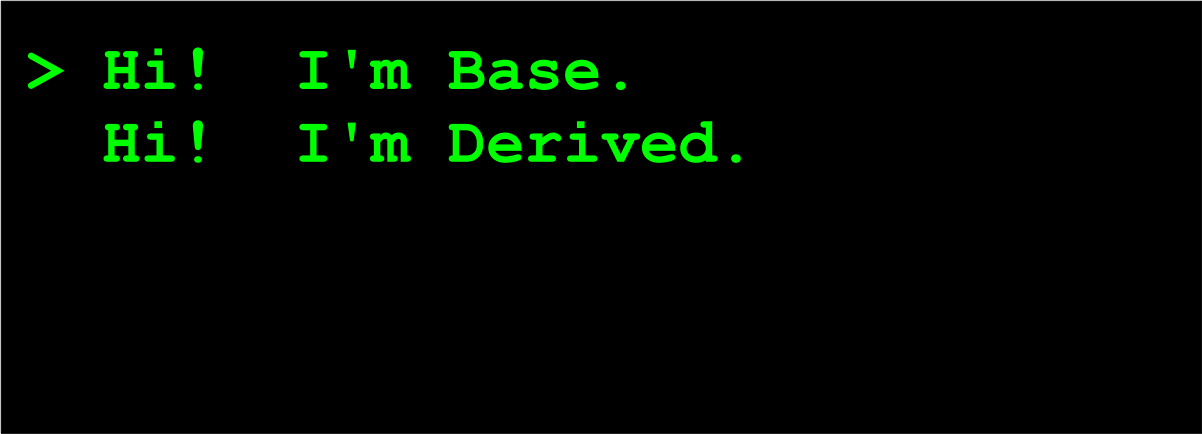
```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```



```
> Hi!  I'm Base.  
Hi!  I'm Derived.
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi! I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi! I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```

```
> Hi! I'm Base.
```

```
Hi! I'm Derived.
```

```
ERROR: Base::missingFunction  
is not defined
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```

```
> Hi!  I'm Base.  
Hi!  I'm Derived.  
ERROR: Base::missingFunction  
is not defined
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```

```
$fnName = "sayHello";  
$b->$fnName();
```

```
> Hi!  I'm Base.  
Hi!  I'm Derived.  
ERROR: Base::missingFunction  
is not defined
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```

```
$fnName = "sayHello";  
$b->$fnName();
```

```
> Hi!  I'm Base.  
Hi!  I'm Derived.  
ERROR: Base::missingFunction  
is not defined
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi!  I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi!  I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```

```
$fnName = "sayHello";  
$b->$fnName();
```

```
> Hi!  I'm Base.  
Hi!  I'm Derived.  
ERROR: Base::missingFunction  
is not defined  
Hi!  I'm Base.
```

Inheritance in PHP

```
class Base {  
    public function sayHello() {  
        echo "Hi! I'm Base.";  
    }  
}
```

```
class Derived extends Base {  
    public function sayHello() {  
        echo "Hi! I'm Derived.";  
    }  
}
```

```
$b = new Base();  
$b->sayHello();
```

```
$d = new Derived();  
$d->sayHello();
```

```
$b->missingFunction();
```

```
$fnName = "sayHello";  
$b->$fnName();
```

```
> Hi! I'm Base.  
Hi! I'm Derived.  
ERROR: Base::missingFunction  
is not defined  
Hi! I'm Base.
```

PHP Inhibits Vtables

- **Call-by-string bypasses the vtable optimization.**
 - Impossible to statically determine contents of any string.
 - Would have to determine index into vtable at runtime.
- **No static type information on objects.**
 - Impossible to statically determine whether a given method exists at all.
- **Plus a few others:**
 - `eval` keyword executes arbitrary PHP code; could introduce new classes or methods.

Inheritance without Vtables

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

```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```

Inheritance without Vtables

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

Code for
Base.sayHi

Code for
Base.clone

Code for
Derived.clone

```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```

Inheritance without Vtables

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

```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```

Code for
Base.sayHi

Code for
Base.clone

Code for
Derived.clone

Vtable*

Base.x

Inheritance without Vtables

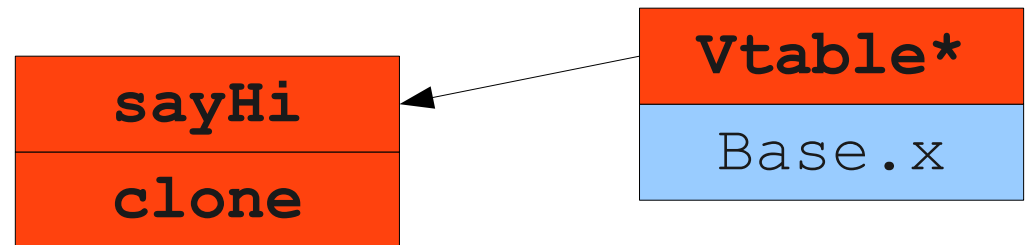
```
class Base {  
    int x;  
    void sayHi() {  
        Print("Hi!");  
    }  
    Base clone() {  
        return new Base;  
    }  
}
```

```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```

Code for
Base.sayHi

Code for
Base.clone

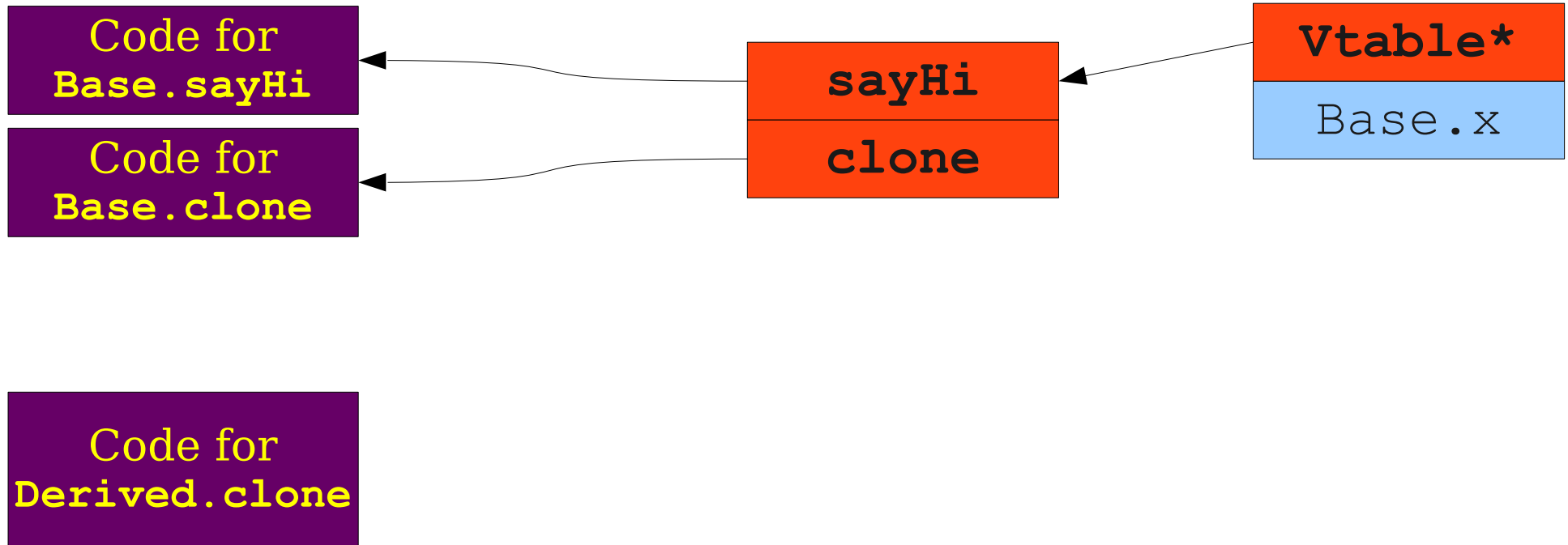
Code for
Derived.clone



Inheritance without Vtables

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

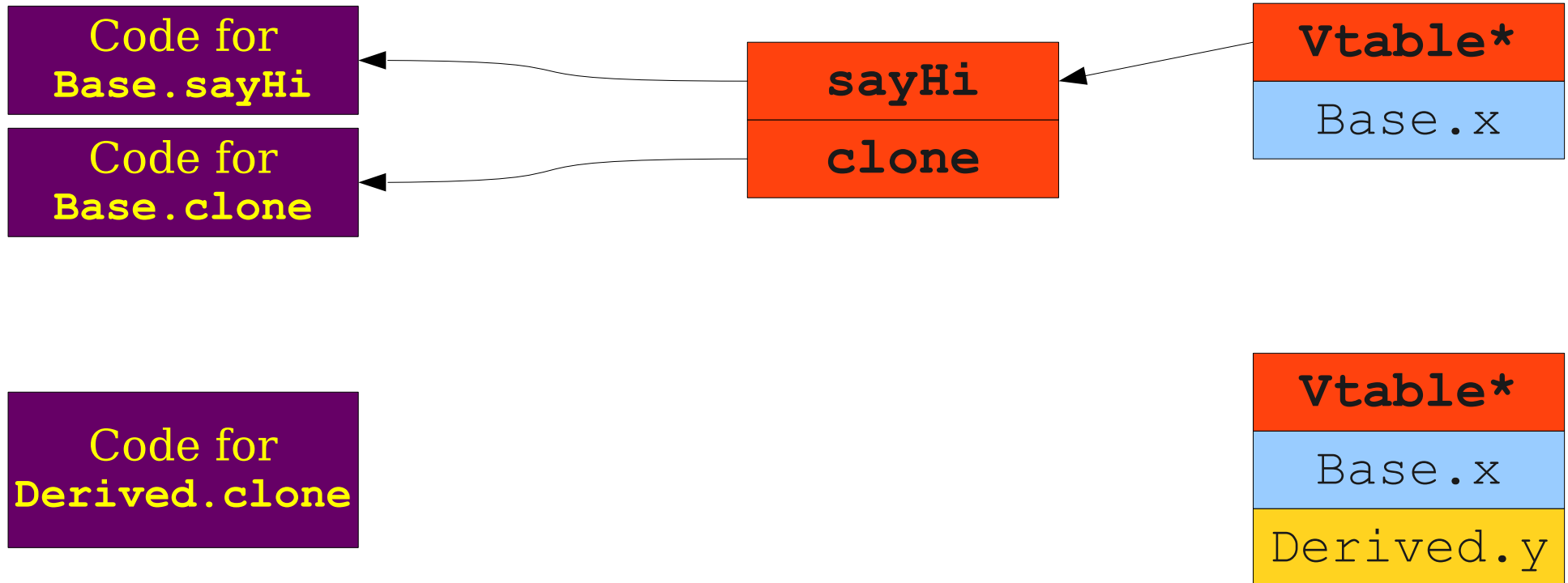
```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```



Inheritance without Vtables

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

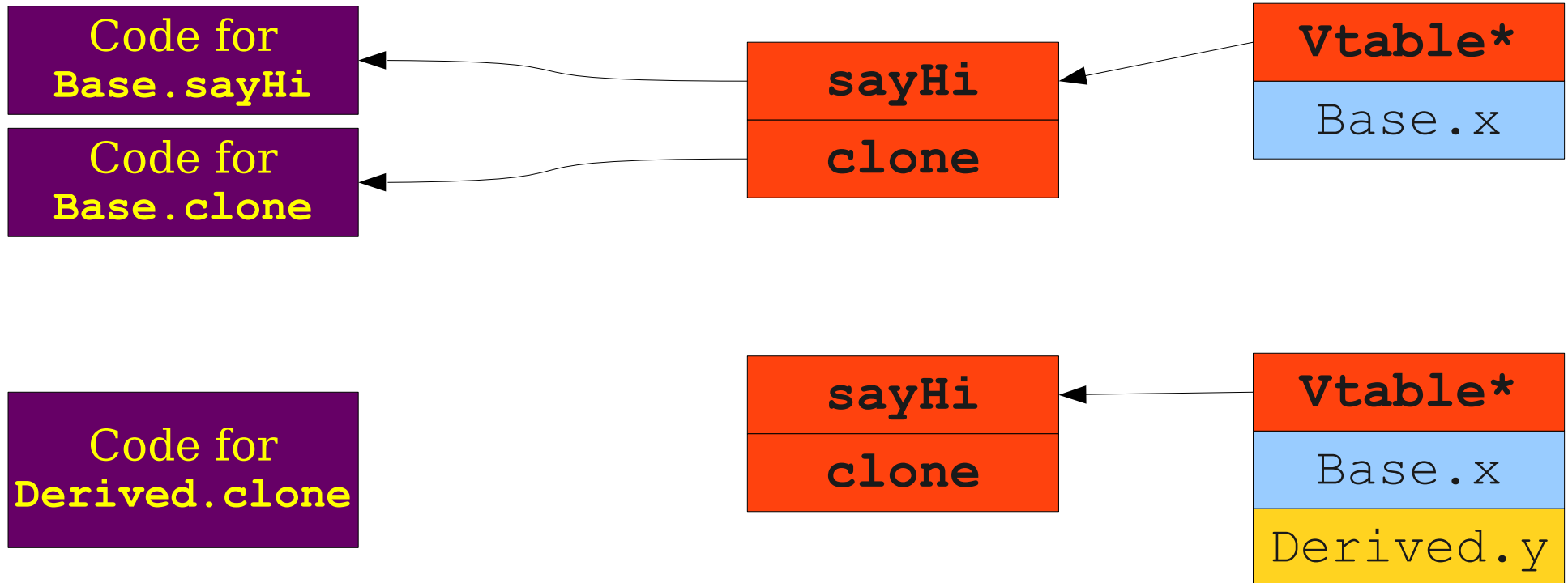
```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```



Inheritance without Vtables

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

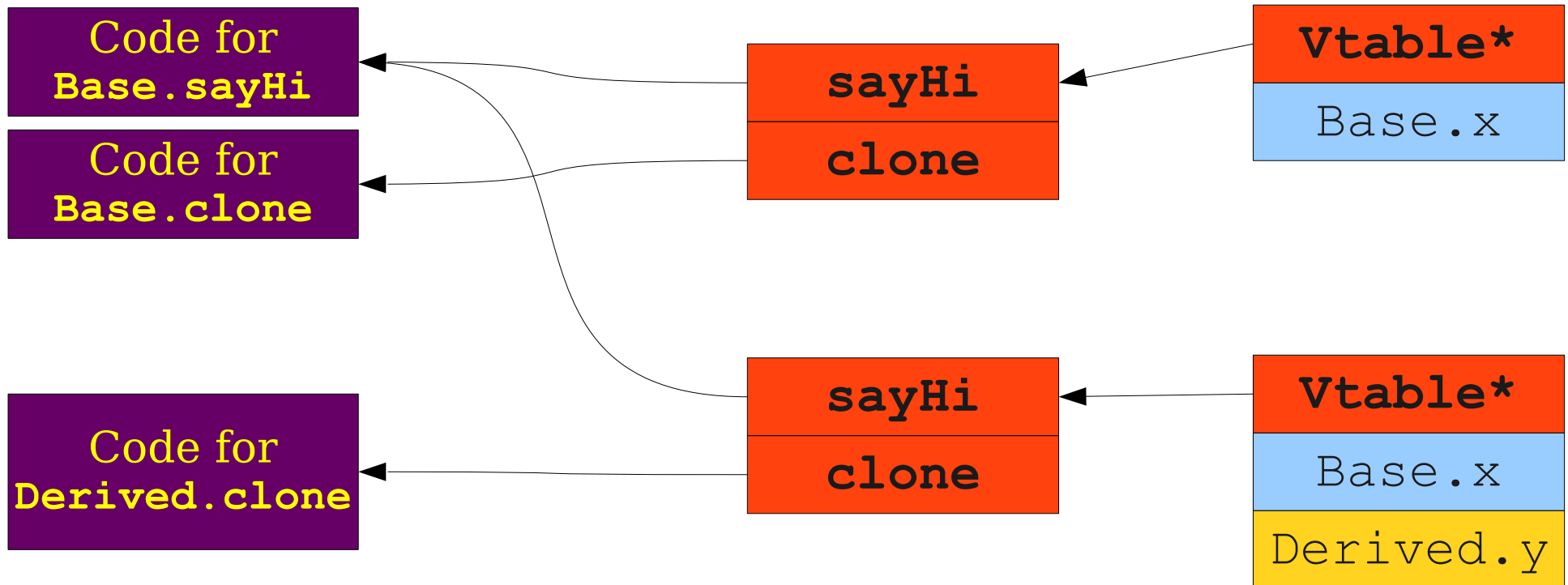
```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```



Inheritance without Vtables

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

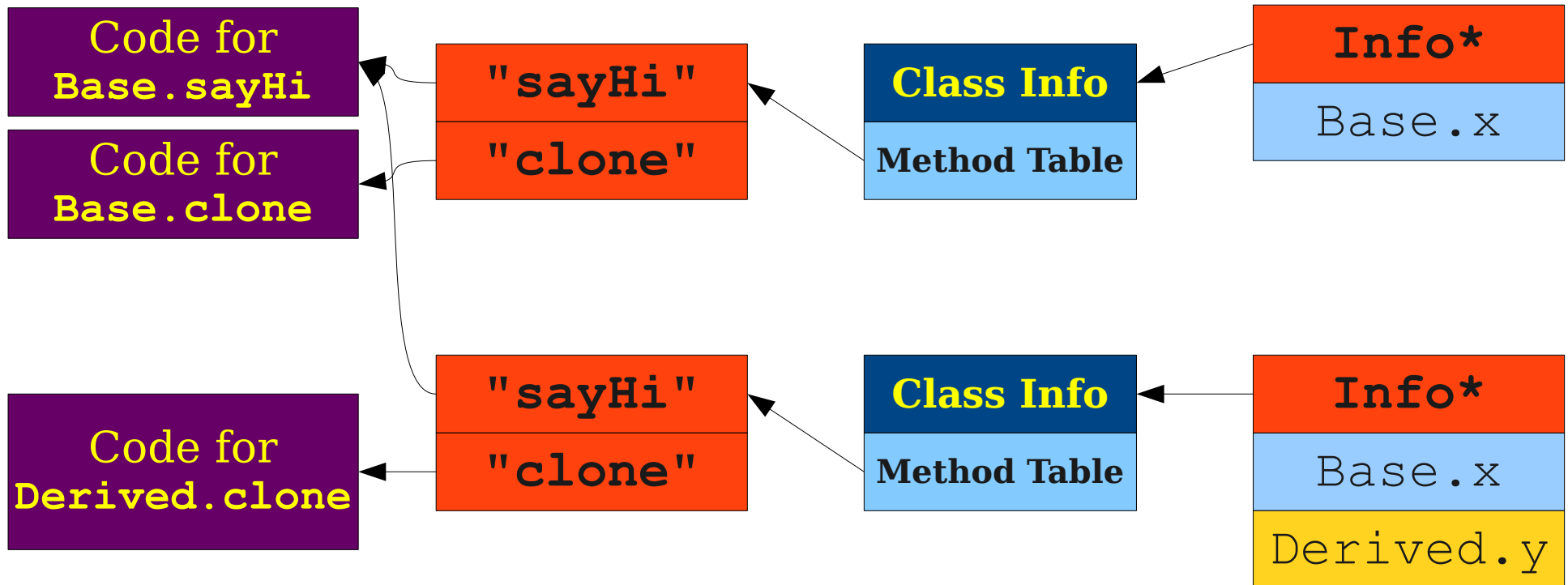
```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```



Inheritance without Vtables

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

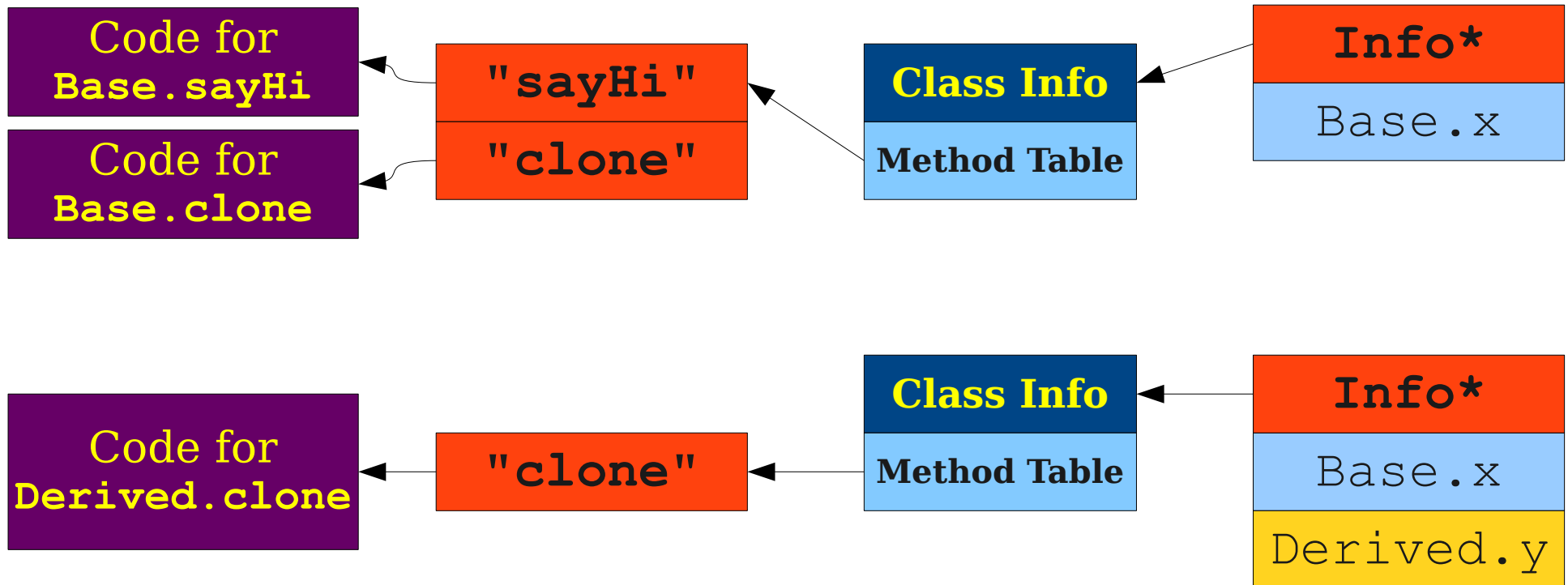
```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```



Inheritance without Vtables

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

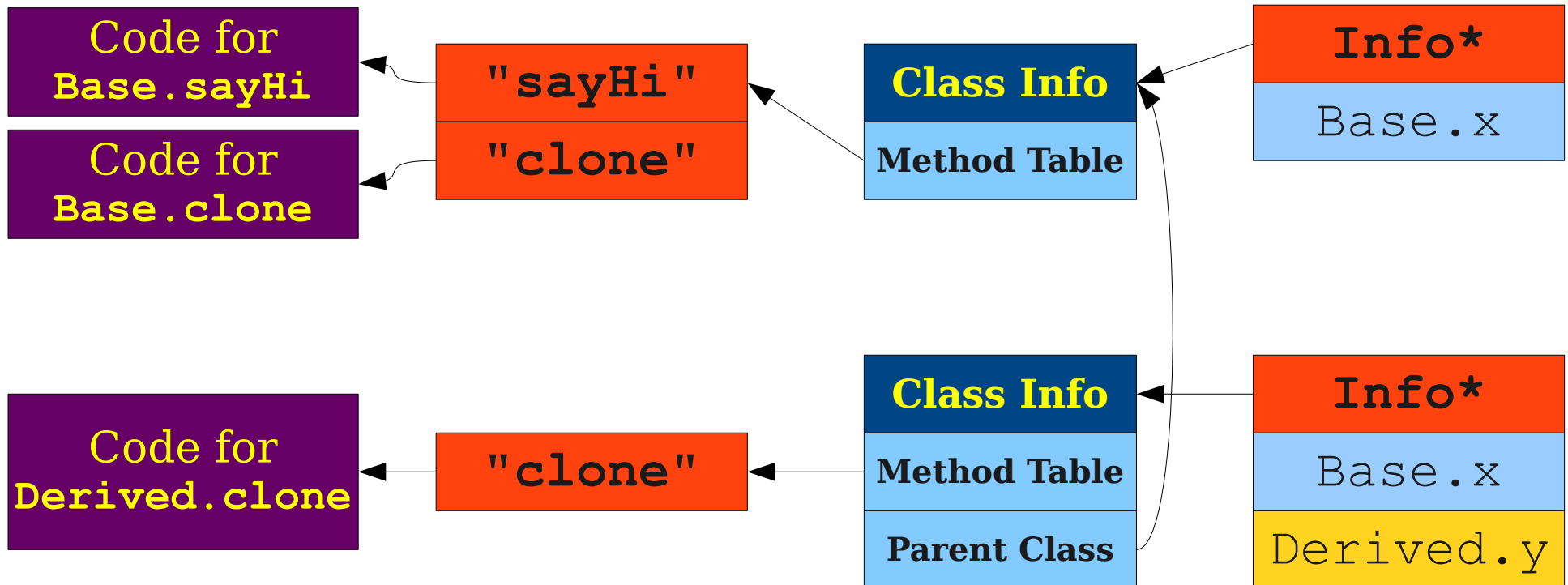
```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```



Inheritance without Vtables

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

```
class Derived extends Base {  
    int y;  
    Derived clone() {  
        return new Derived;  
    }  
}
```



A General Inheritance Framework

- Each object stores a pointer to a descriptor for its class.
- Each class descriptor stores
 - A pointer to the base class descriptor(s).
 - A pointer to a method lookup table.
- To invoke a method:
 - Follow the pointer to the method table.
 - If the method exists, call it.
 - Otherwise, navigate to the base class and repeat.
- This is slow but can be optimized in many cases; we'll see this later.

Vtables and Interfaces

```
interface Engine {
    void vroom();
}
interface Visible {
    void draw();
}
class PaintedEngine implements Engine, Visible {
    void vroom() { /* ... */ }
    void draw() { /* ... */ }
}
class JetEngine implements Engine {
    void vroom() { /* ... */ }
}
class Paint implements Visible {
    void draw() { /* ... */ }
}
```

```
Engine e1  = new PaintedEngine;
Engine e2  = new JetEngine;
e1.vroom();
e2.vroom();
Visible v1 = new PaintedEngine;
Visible v2 = new Paint;
v1.draw();
v2.draw();
```

Vtables and Interfaces

```
interface Engine {  
    void vroom();  
}  
interface Visible {  
    void draw();  
}  
class PaintedEngine implements Engine, Visible {  
    void vroom() { /* ... */ }  
    void draw() { /* ... */ }  
}  
class JetEngine implements Engine {  
    void vroom() { /* ... */ }  
}  
class Paint implements Visible {  
    void draw() { /* ... */ }  
}
```

```
Engine e1 = new PaintedEngine;  
Engine e2 = new JetEngine;  
e1.vroom();  
e2.vroom();  
Visible v1 = new PaintedEngine;  
Visible v2 = new Paint;  
v1.draw();  
v2.draw();
```

PaintedEngine vtable

vroom	draw
-------	------

Vtables and Interfaces

```
interface Engine {  
    void vroom();  
}  
interface Visible {  
    void draw();  
}  
class PaintedEngine implements Engine, Visible {  
    void vroom() { /* ... */ }  
    void draw() { /* ... */ }  
}  
class JetEngine implements Engine {  
    void vroom() { /* ... */ }  
}  
class Paint implements Visible {  
    void draw() { /* ... */ }  
}
```

```
Engine e1 = new PaintedEngine;  
Engine e2 = new JetEngine;  
e1.vroom();  
e2.vroom();  
Visible v1 = new PaintedEngine;  
Visible v2 = new Paint;  
v1.draw();  
v2.draw();
```

PaintedEngine vtable

vroom

draw

JetEngine vtable

vroom

Vtables and Interfaces

```
interface Engine {  
    void vroom();  
}  
interface Visible {  
    void draw();  
}  
class PaintedEngine implements Engine, Visible {  
    void vroom() { /* ... */ }  
    void draw() { /* ... */ }  
}  
class JetEngine implements Engine {  
    void vroom() { /* ... */ }  
}  
class Paint implements Visible {  
    void draw() { /* ... */ }  
}
```

```
Engine e1 = new PaintedEngine;  
Engine e2 = new JetEngine;  
e1.vroom();  
e2.vroom();  
Visible v1 = new PaintedEngine;  
Visible v2 = new Paint;  
v1.draw();  
v2.draw();
```

PaintedEngine vtable

vroom	draw
-------	------

JetEngine vtable

vroom

Paint vtable

(empty)	draw
---------	------

Interfaces with Vtables

- Interfaces complicate vtable layouts because they require interface methods to have consistent positions across all vtables.
- This can fill vtables with useless entries.
- For this reason, interfaces are typically not implemented using pure vtables.

Interfaces via String Lookup

- Idea: A hybrid approach.
- Use vtables for standard (non-interface) dispatch.
- Use the more general, string-based lookup for interfaces.

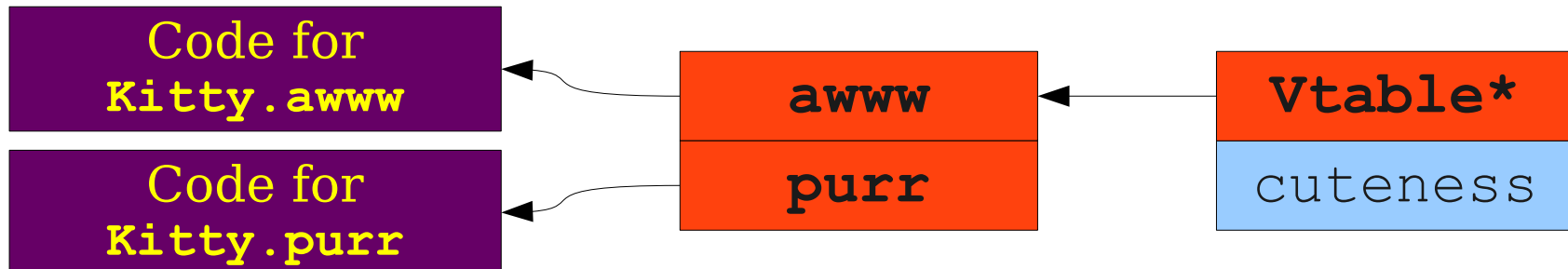
Object Layout with Interfaces

```
class Kitty implements Adorable {  
    int cuteness;  
    void awww() {  
        Print("Meow");  
    }  
    void purr() {  
        Print("Purr");  
    }  
}  
  
interface Adorable {  
    void awww();  
}
```

Object Layout with Interfaces

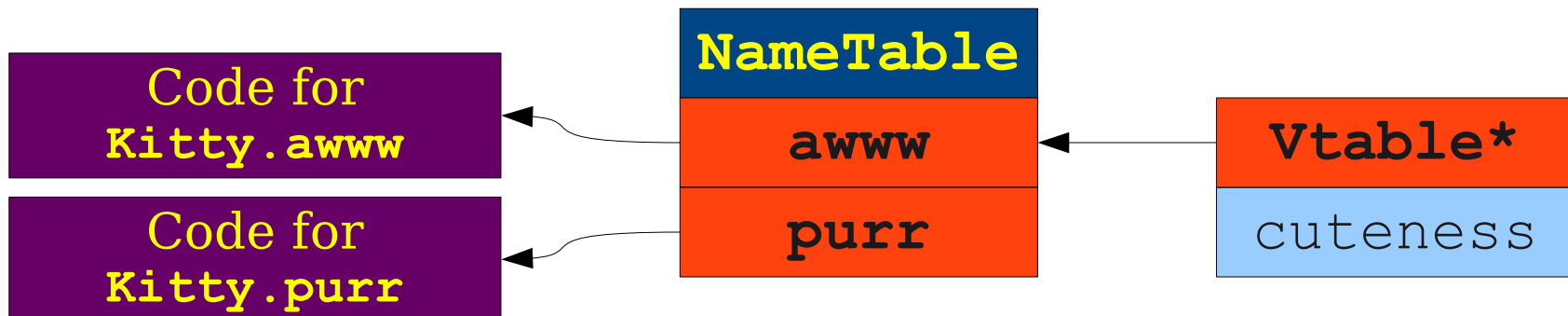
```
class Kitty implements Adorable {  
    int cuteness;  
    void awww() {  
        Print("Meow");  
    }  
    void purr() {  
        Print("Purr");  
    }  
}
```

```
interface Adorable {  
    void awww();  
}
```



Object Layout with Interfaces

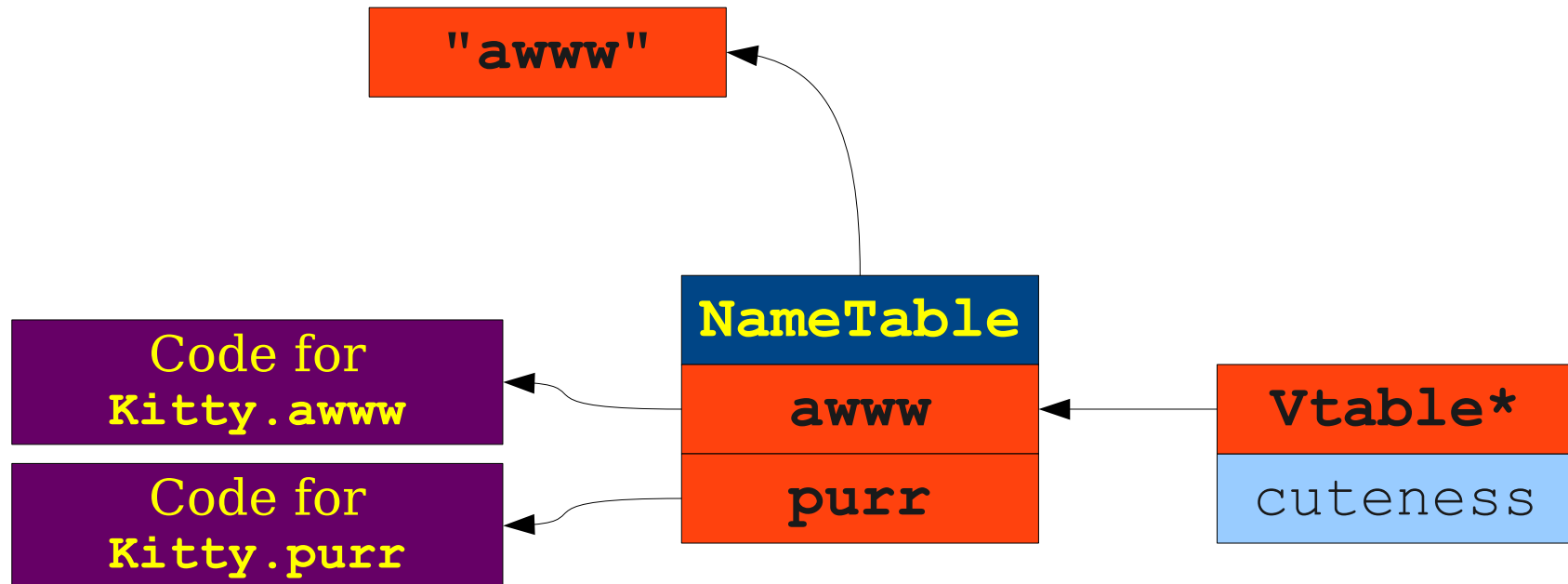
```
class Kitty implements Adorable {  
    int cuteness;  
    void awww() {  
        Print("Meow");  
    }  
    void purr() {  
        Print("Purr");  
    }  
}  
  
interface Adorable {  
    void awww();  
}
```



Object Layout with Interfaces

```
class Kitty implements Adorable {  
    int cuteness;  
    void awww() {  
        Print("Meow");  
    }  
    void purr() {  
        Print("Purr");  
    }  
}
```

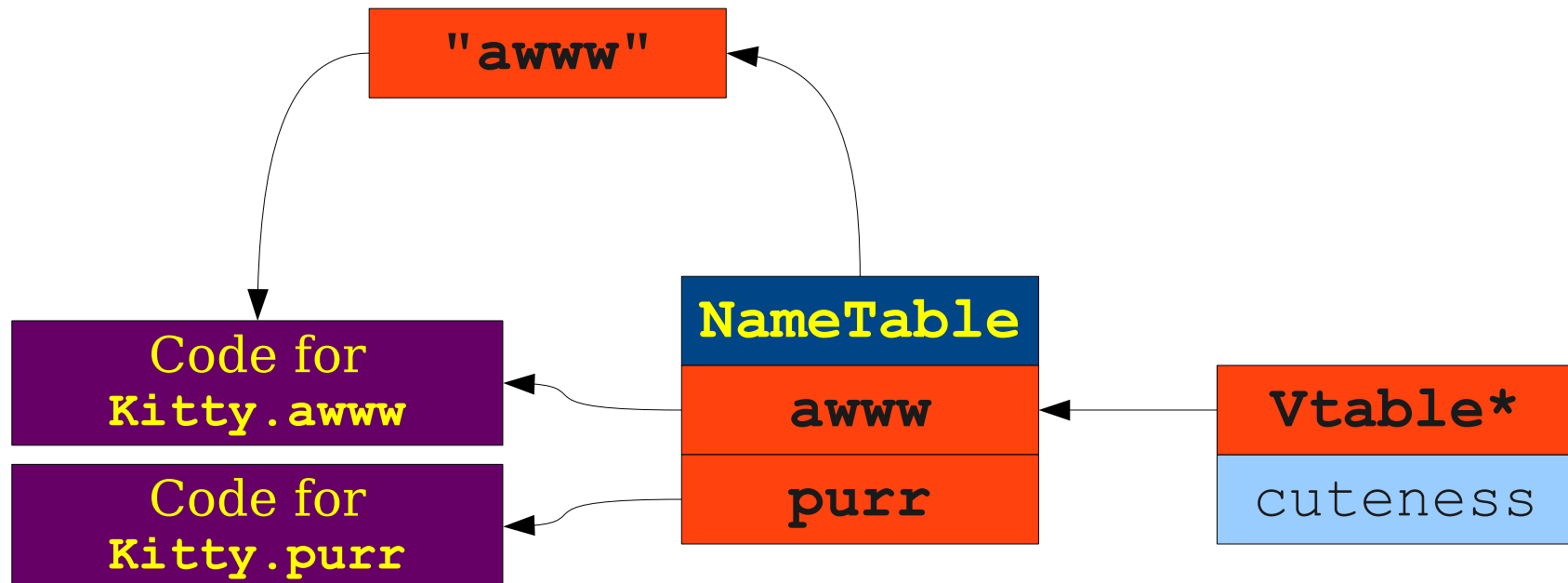
```
interface Adorable {  
    void awww();  
}
```



Object Layout with Interfaces

```
class Kitty implements Adorable {  
    int cuteness;  
    void awww() {  
        Print("Meow");  
    }  
    void purr() {  
        Print("Purr");  
    }  
}
```

```
interface Adorable {  
    void awww();  
}
```



Analysis of the Approach

- Dynamic dispatch through object types still $O(1)$.
- Interface dispatches take $O(Mn)$, where M is the number of methods and n is the length of the method name.
- Can easily speed up to $O(n)$ expected by replacing a list of strings with a hash table.

Implementing Dynamic Type Checks

Dynamic Type Checks

- Many languages require some sort of dynamic type checking.
 - Java's `instanceof`, C++'s `dynamic_cast`, any dynamically-typed language.
- May want to determine whether the dynamic type is *convertible* to some other type, not whether the type is *equal*.
- How can we implement this?

A Pretty Good Approach

```
class A {  
    void f() {}  
}
```

```
class B extends A {  
    void f() {}  
}
```

```
class C extends A {  
    void f() {}  
}
```

```
class D extends B {  
    void f() {}  
}
```

```
class E extends C {  
    void f() {}  
}
```

A Pretty Good Approach

```
class A {  
    void f() {}  
}
```

A.f

```
class B extends A {  
    void f() {}  
}
```

B.f

```
class C extends A {  
    void f() {}  
}
```

C.f

```
class D extends B {  
    void f() {}  
}
```

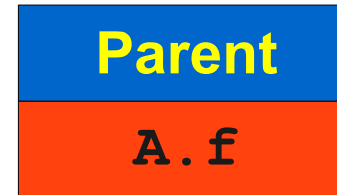
D.f

```
class E extends C {  
    void f() {}  
}
```

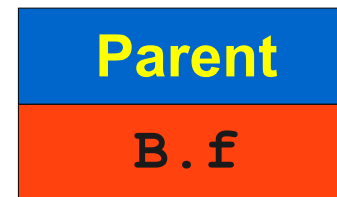
E.f

A Pretty Good Approach

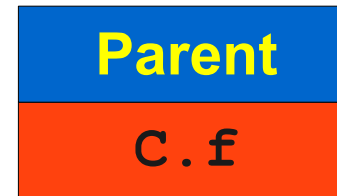
```
class A {  
    void f() {}  
}
```



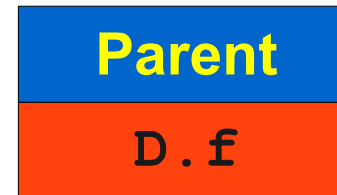
```
class B extends A {  
    void f() {}  
}
```



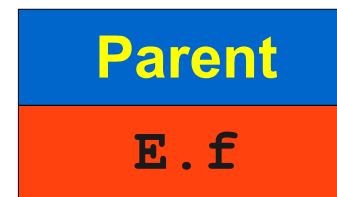
```
class C extends A {  
    void f() {}  
}
```



```
class D extends B {  
    void f() {}  
}
```

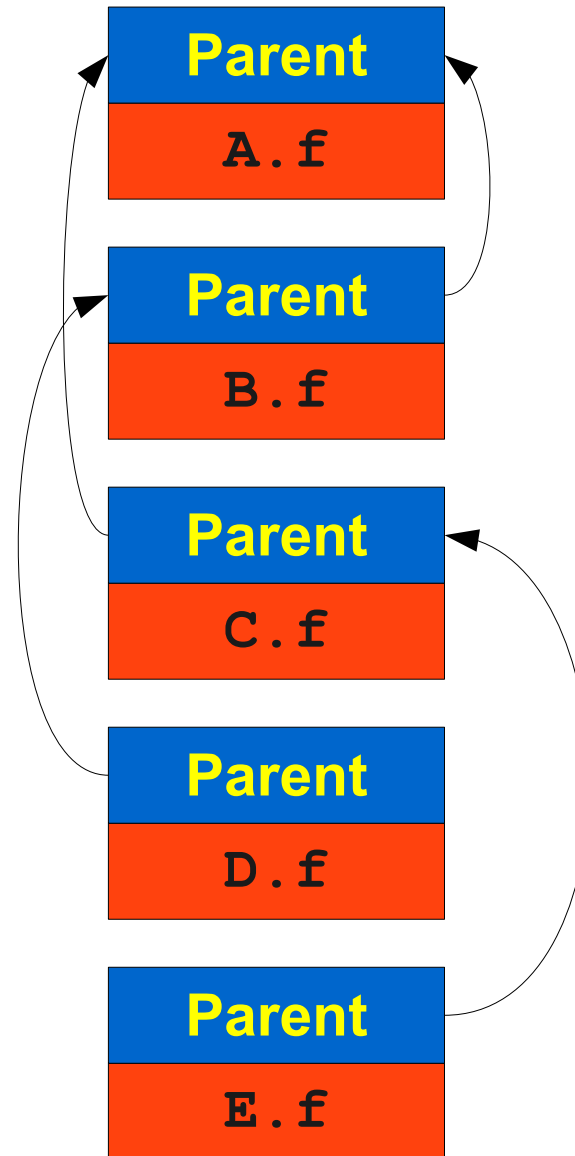


```
class E extends C {  
    void f() {}  
}
```



A Pretty Good Approach

```
class A {  
    void f() {}  
}  
  
class B extends A {  
    void f() {}  
}  
  
class C extends A {  
    void f() {}  
}  
  
class D extends B {  
    void f() {}  
}  
  
class E extends C {  
    void f() {}  
}
```



Simple Dynamic Type Checking

- Have each object's vtable store a pointer to its base class.
- To check if an object is convertible to type S at runtime, follow the pointers embedded in the object's vtable upward until we find S or reach a type with no parent.
- Runtime is $O(d)$, where d is the depth of the class in the hierarchy.

Next Time

- **Three-Address Code IR.**
- **IR Generation.**