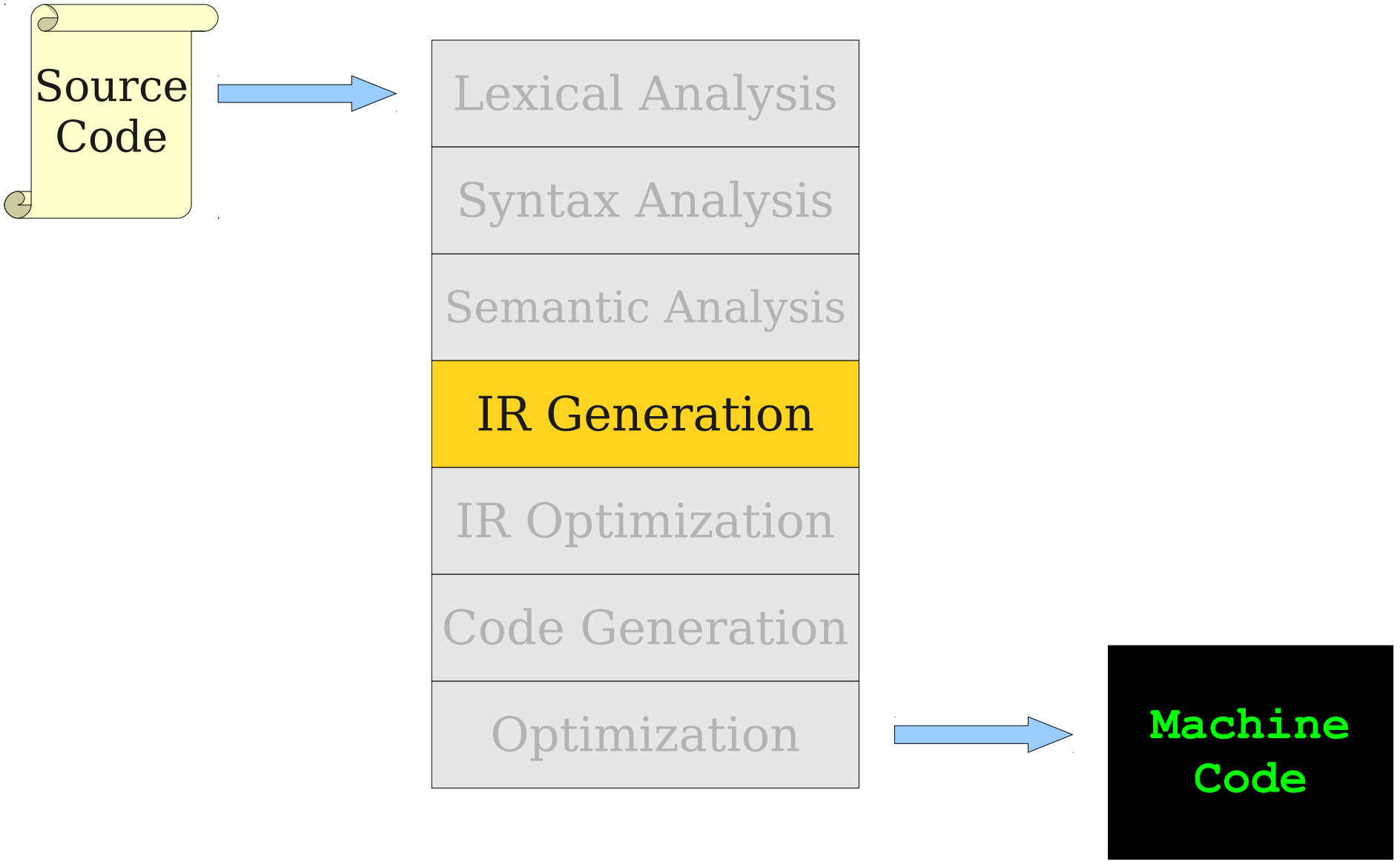
Runtime Environments
Part II

Announcements

- Programming Project 3 Checkpoint due tonight at 11:59PM.
  - **No late submissions accepted**.
  - Ask questions on Piazza!
  - Stop by office hours!
  - Email the staff list!
- Midterm graded; will be returned at end of class.
  - Available outside Gates 178 afterwards.

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

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.

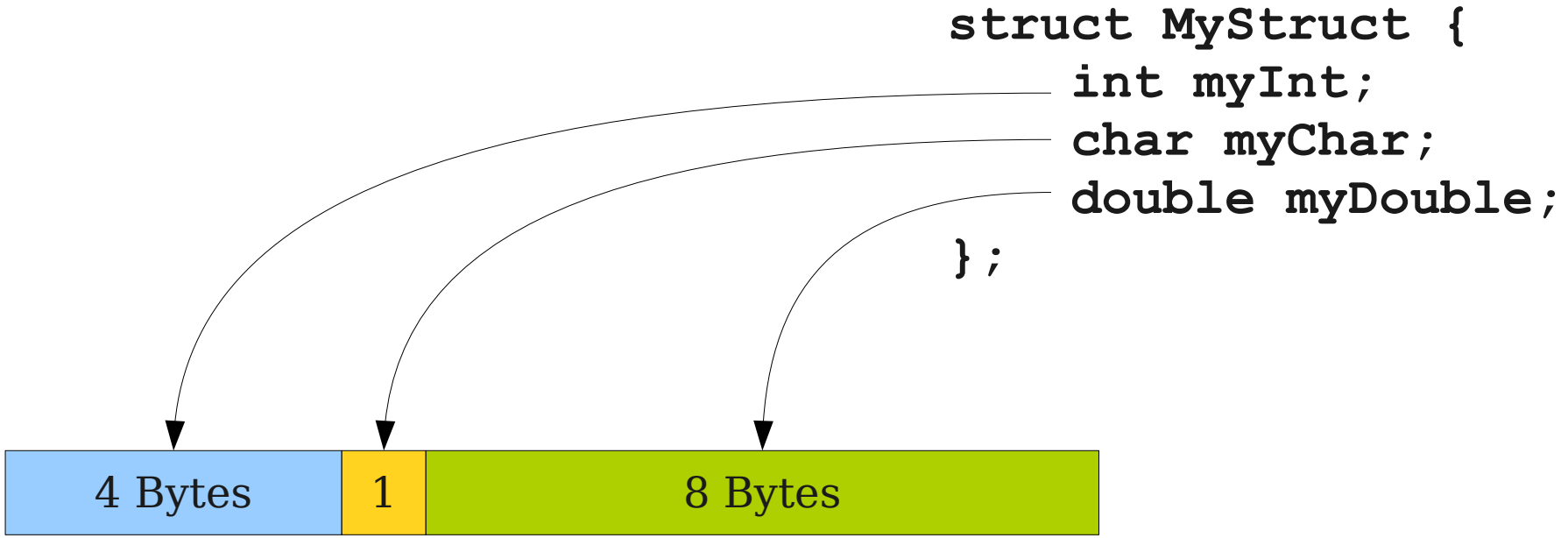
Encoding C-Style **`structs`**

- 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;
};
```

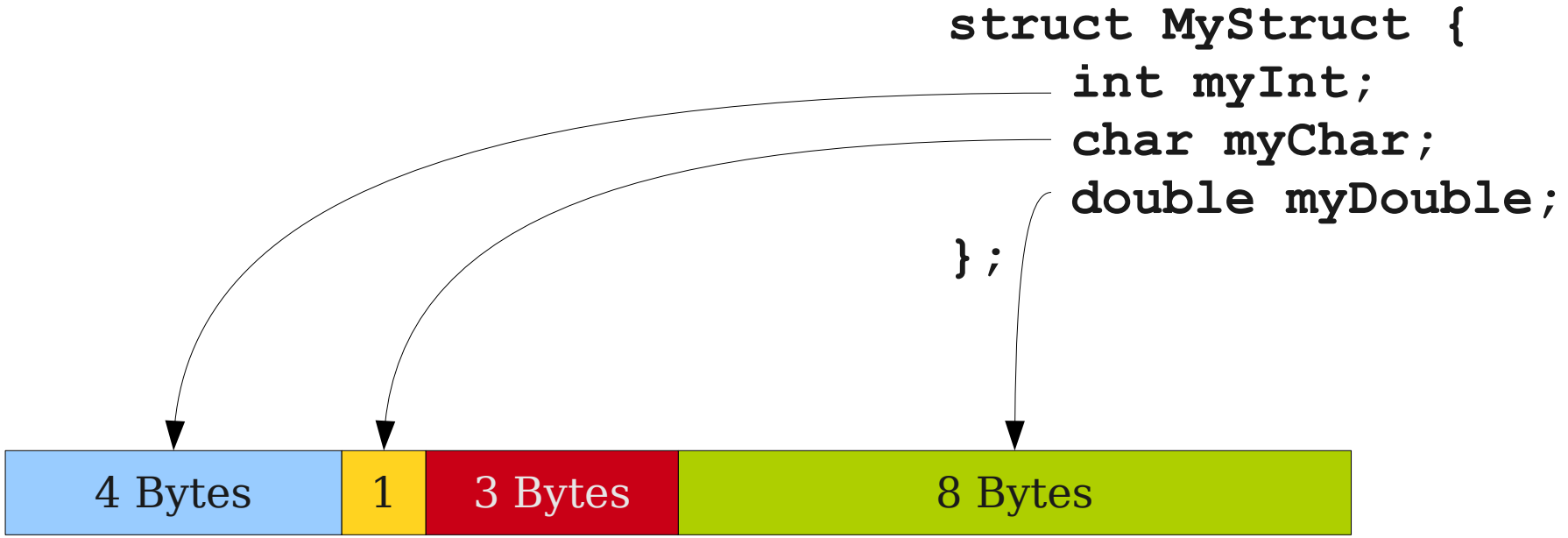
Encoding C-Style **`structs`**

- 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

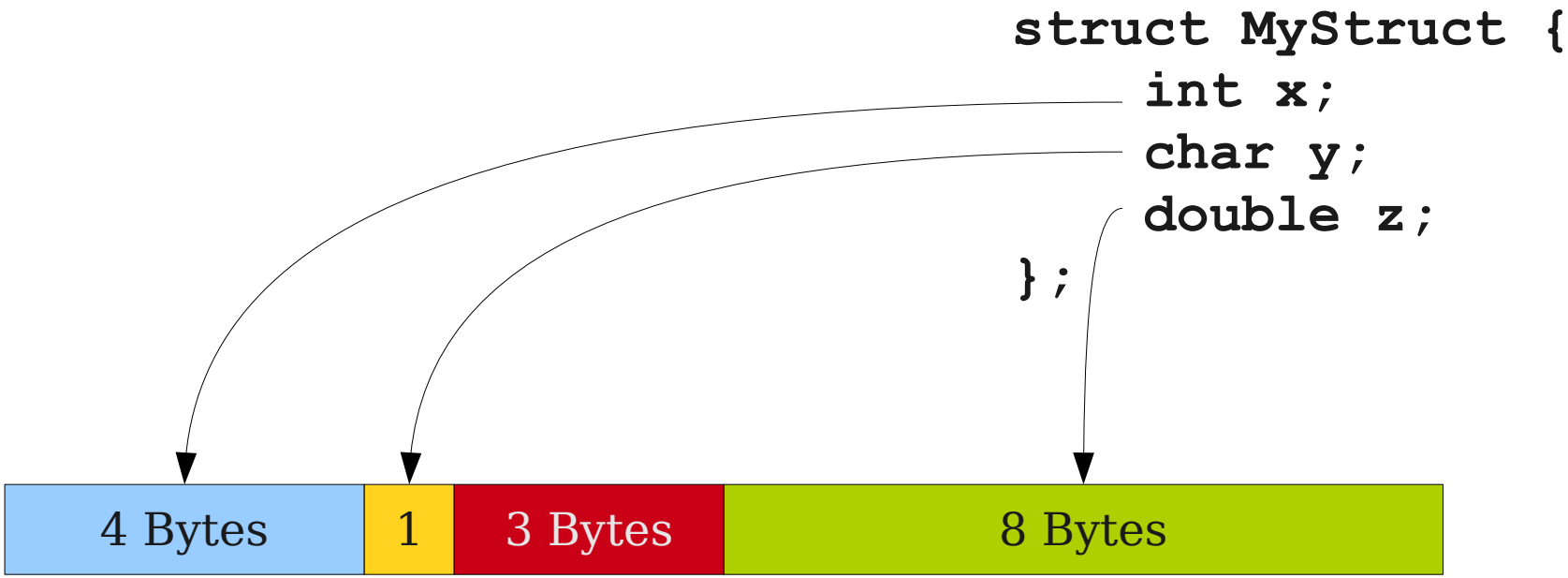
- Once an object is laid out in memory, it's just a series of bytes.
- How do we know where to look to find a particular field?

Accessing Fields

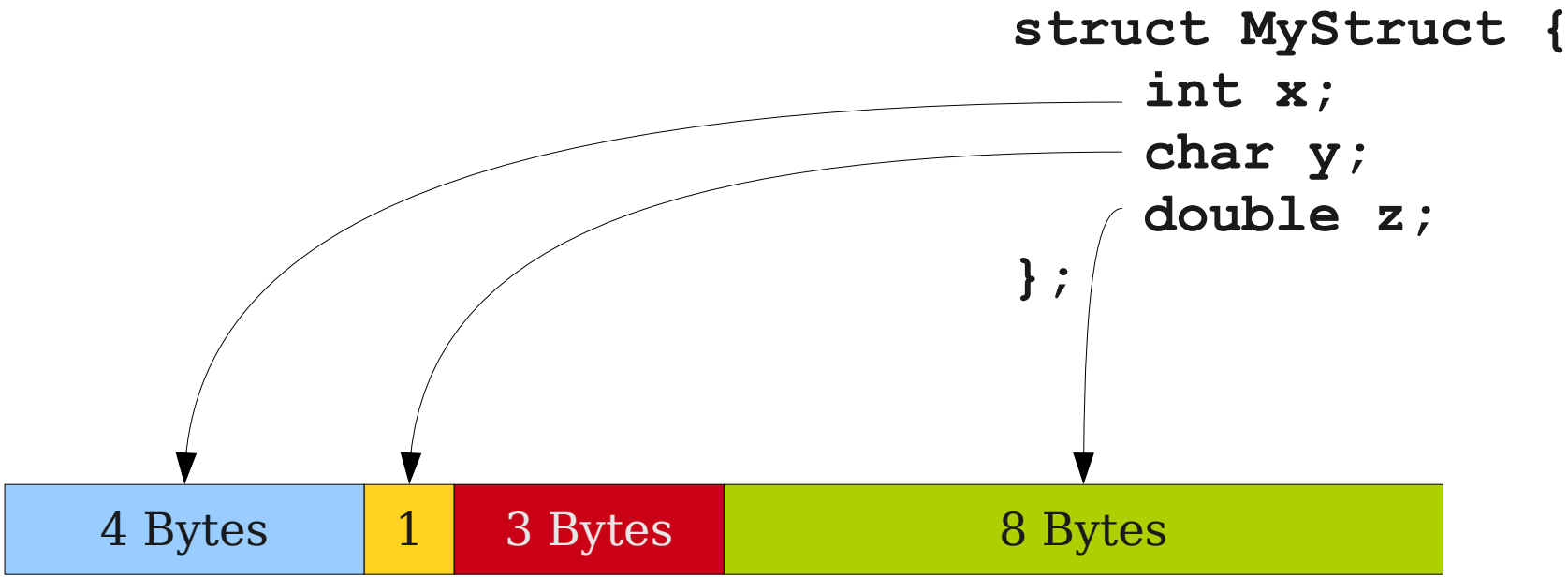
- Once an object is laid out in memory, it's just a series of bytes.
- How do we know where to look to find a particular field?

- 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

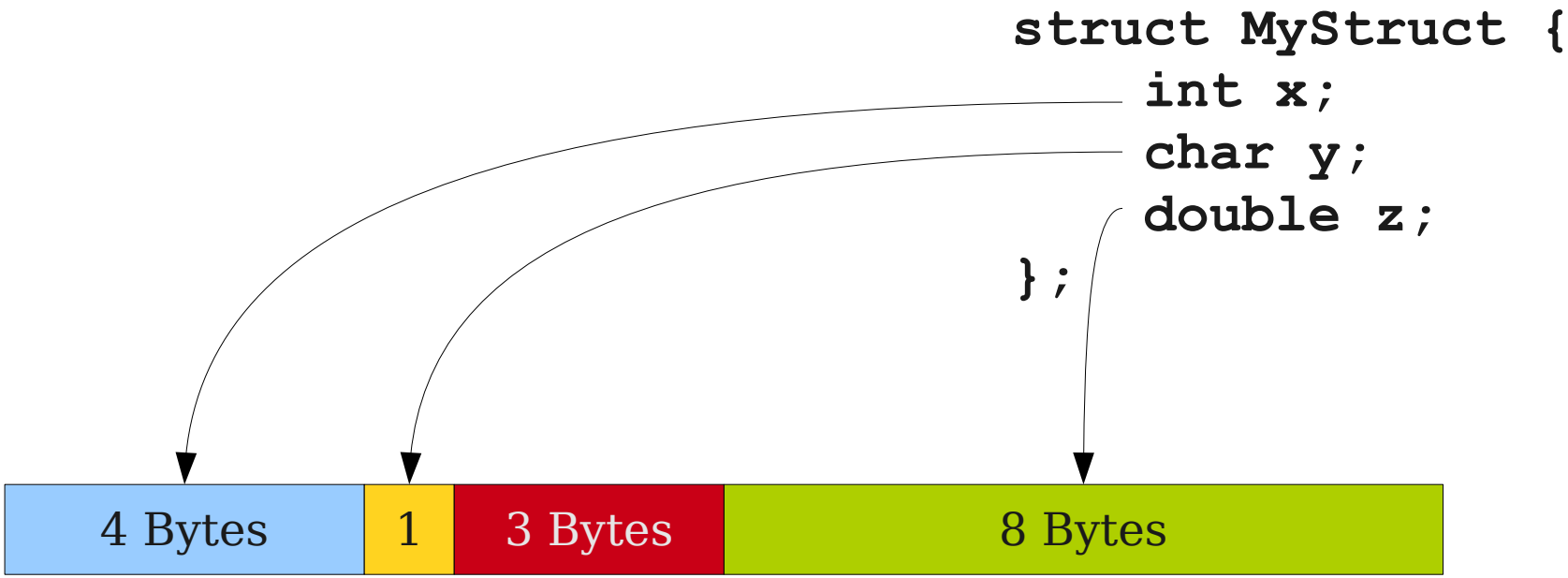


Field Lookup




```
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?

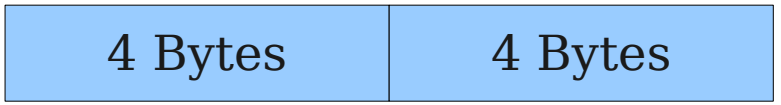
Memory Layouts with Inheritance

Memory Layouts with Inheritance

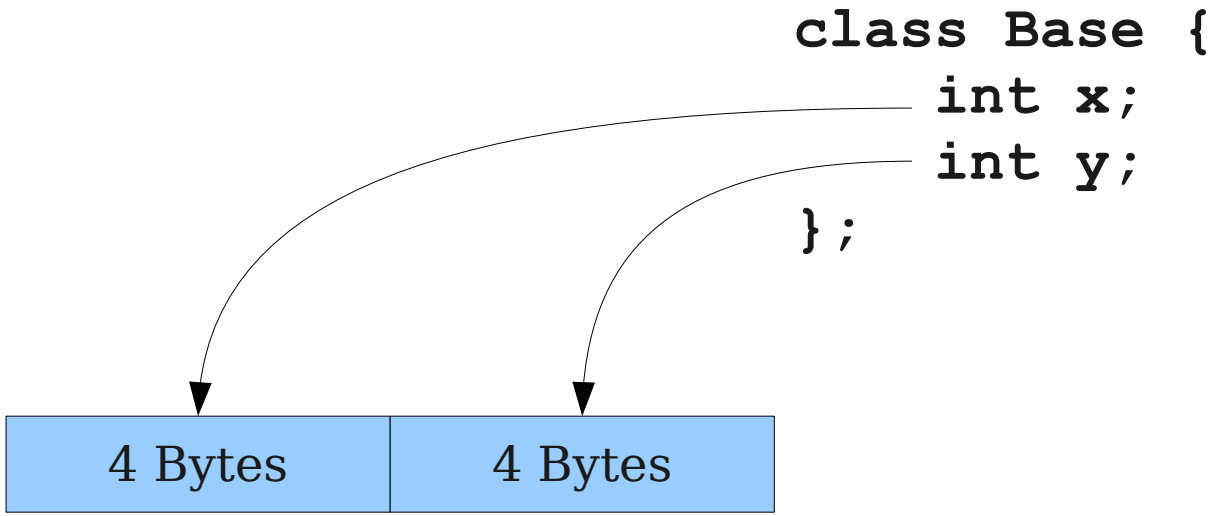
```
class Base {
    int x;
    int y;
};
```

Memory Layouts with Inheritance

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

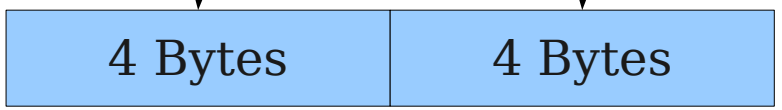


Memory Layouts with Inheritance



Memory Layouts with Inheritance

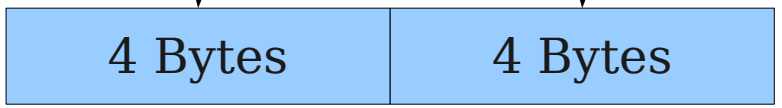
```
class Base {
    int x;
    int y;
};
```




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

Memory Layouts with Inheritance

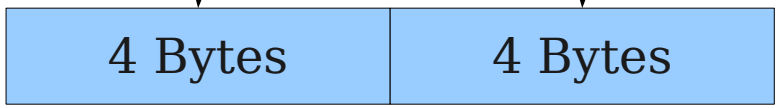
```
class Base {
    int x;
    int y;
};
```



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

Memory Layouts with Inheritance

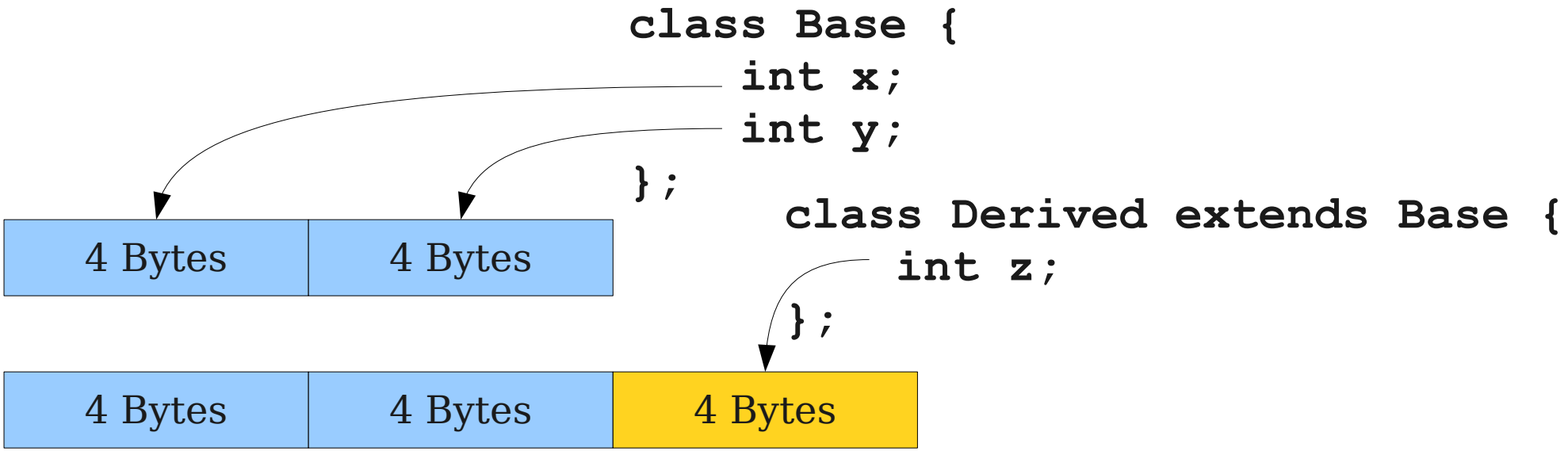
```
class Base {
    int x;
    int y;
};
```



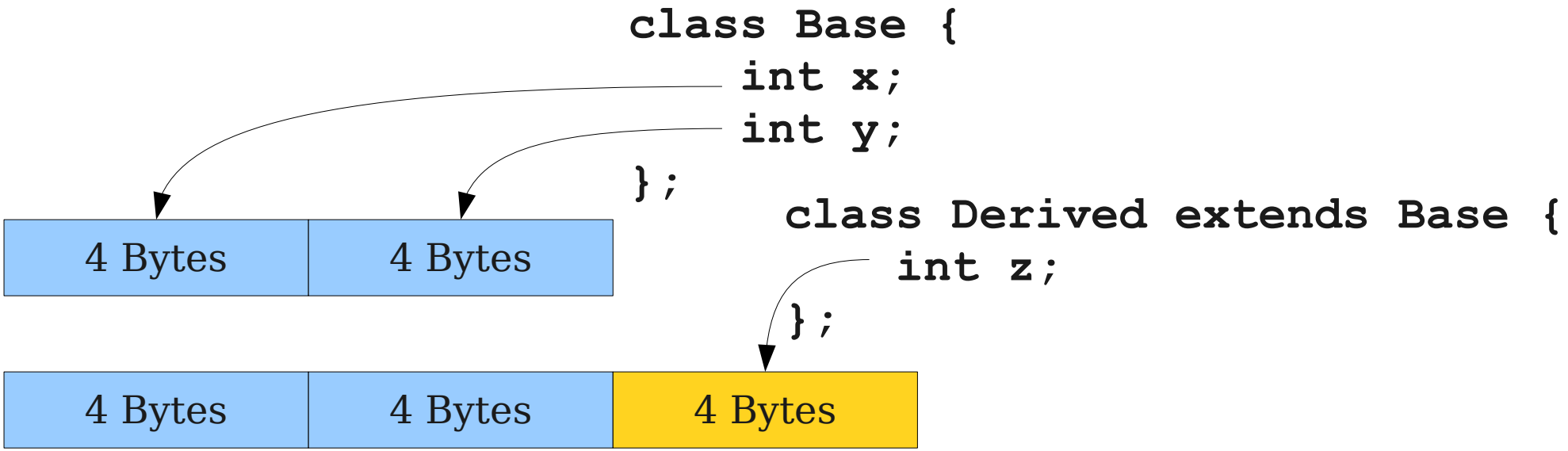
```
class Derived extends Base {
    int z;
};
```

Field Lookup With Inheritance

Field Lookup With Inheritance

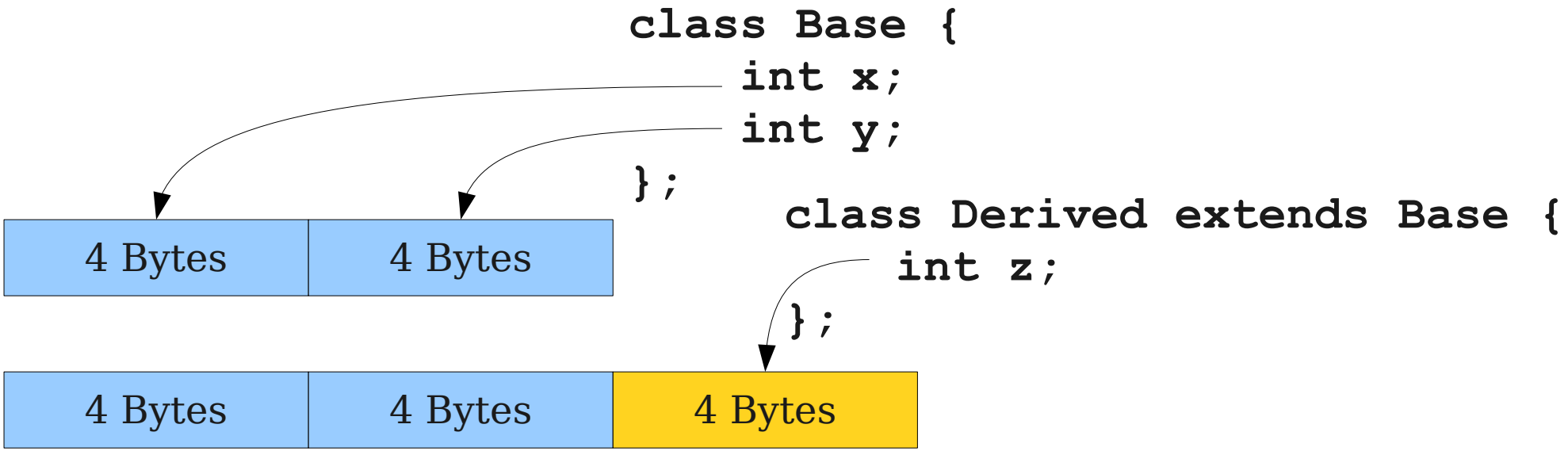


Field Lookup With Inheritance



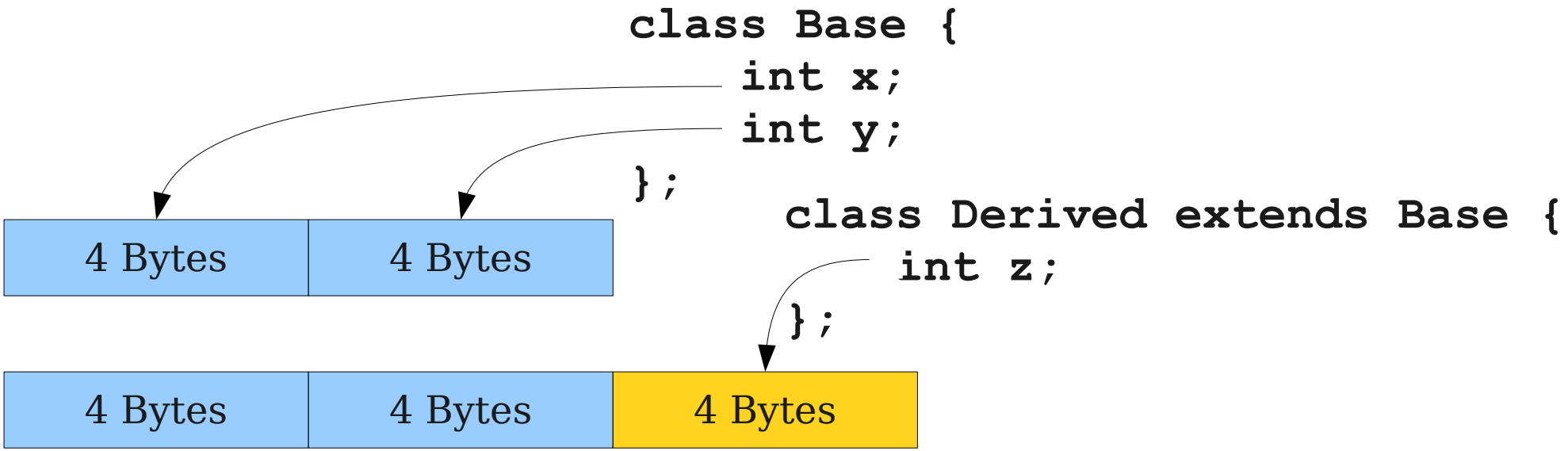

```
Base ms = new Base;
ms.x = 137;
ms.y = 42;
```

Field Lookup With Inheritance



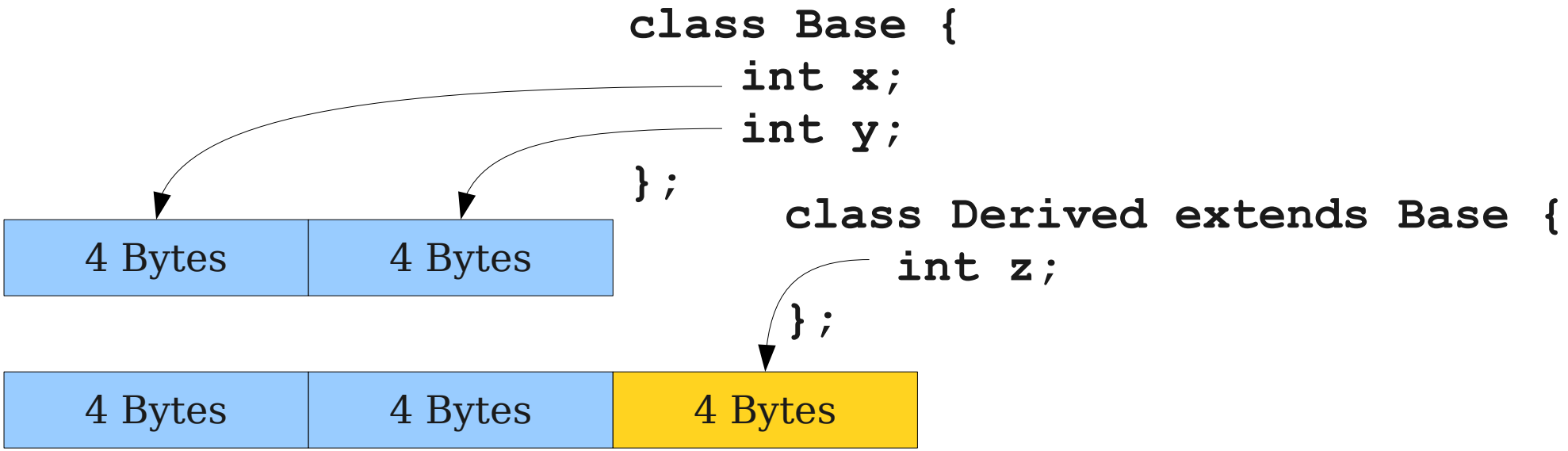

```
Base ms = new Base;
ms.x = 137;     store 137 0 bytes after ms
ms.y = 42;      store 42  4 bytes after ms
```

Field Lookup With Inheritance



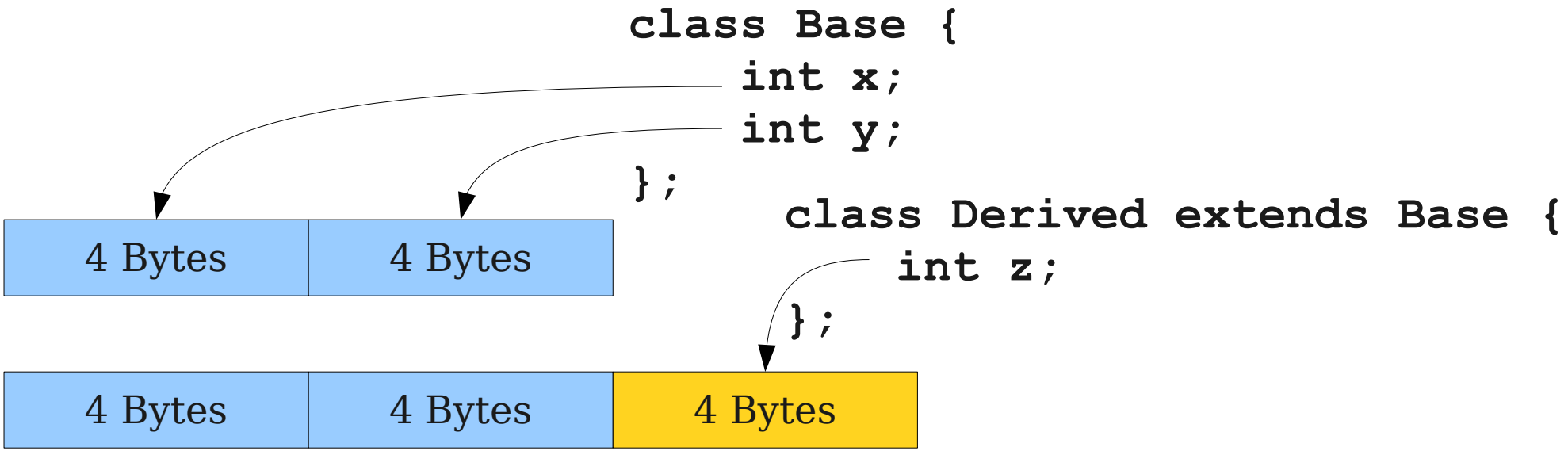

```
Base ms = new Derived;
ms.x = 137;
ms.y = 42;
```

Field Lookup With Inheritance



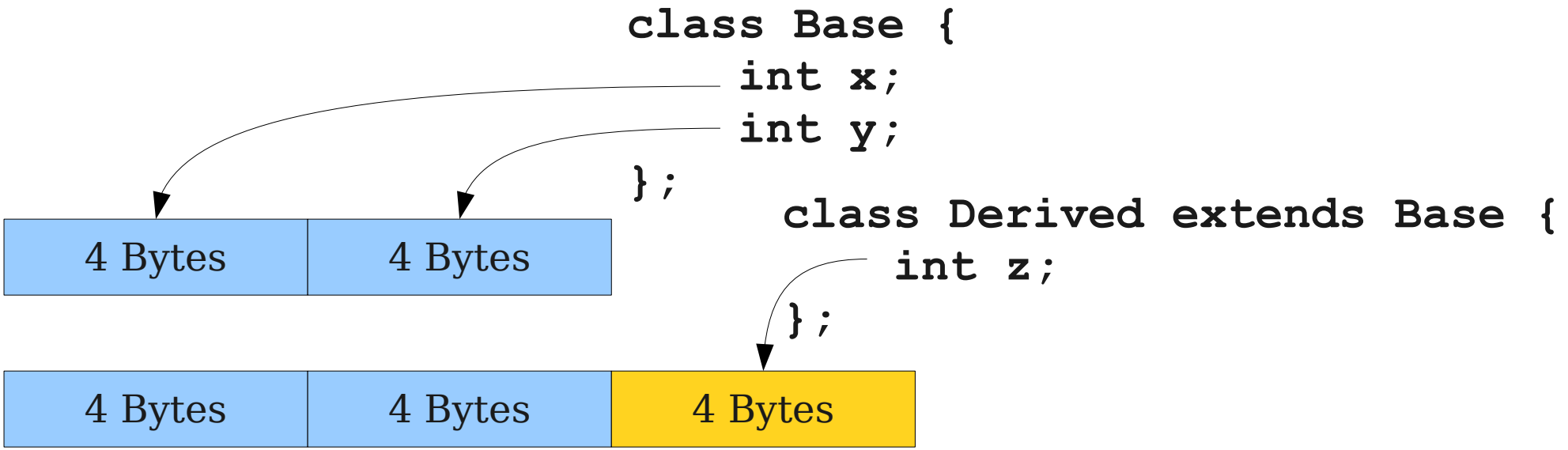

```
Base ms = new Derived;
ms.x = 137;
ms.y = 42;
```

Field Lookup With Inheritance




```
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.
```

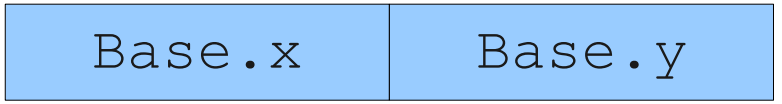
Analyzing our Approach

- 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

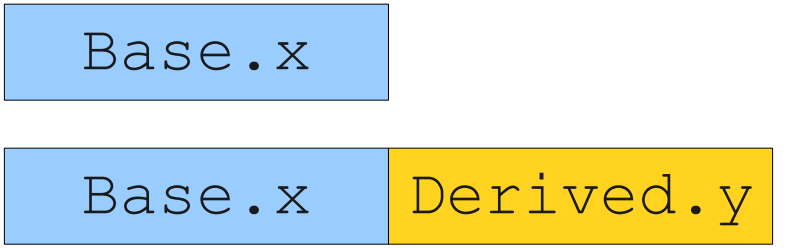
Code for **Base.sayHi**

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");
     }
}
```



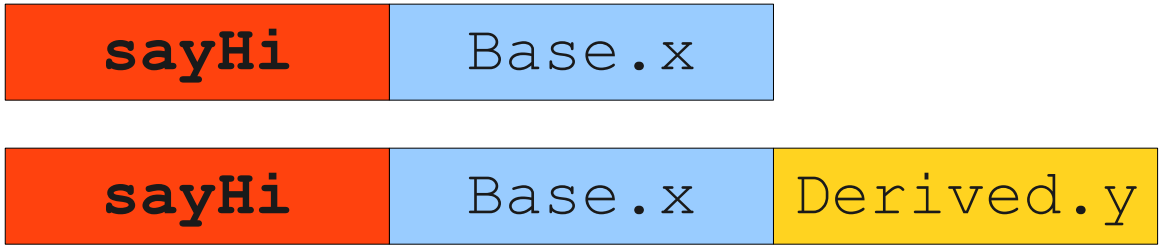
Code for **Base.sayHi**

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");
     }
}
```



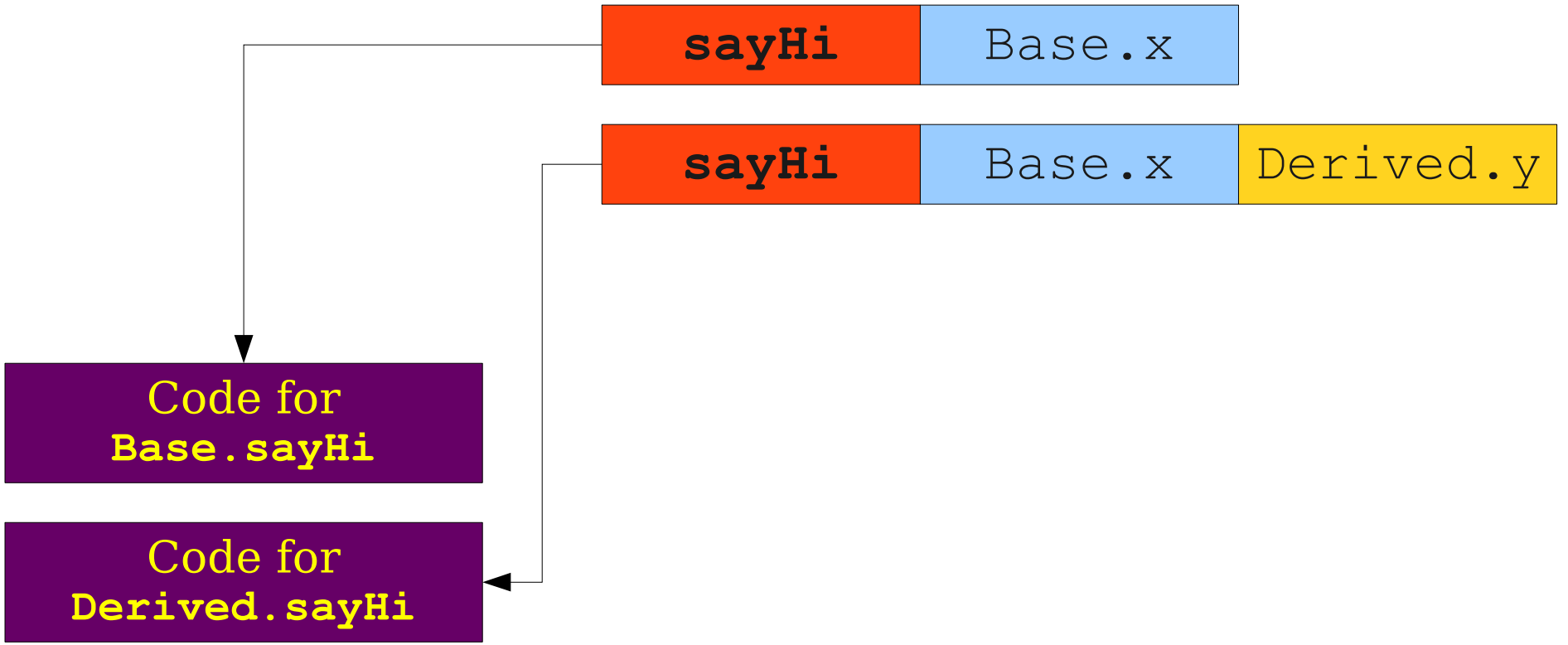
Code for **Base.sayHi**

Code for **Derived.sayHi**

Virtual Function Tables

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

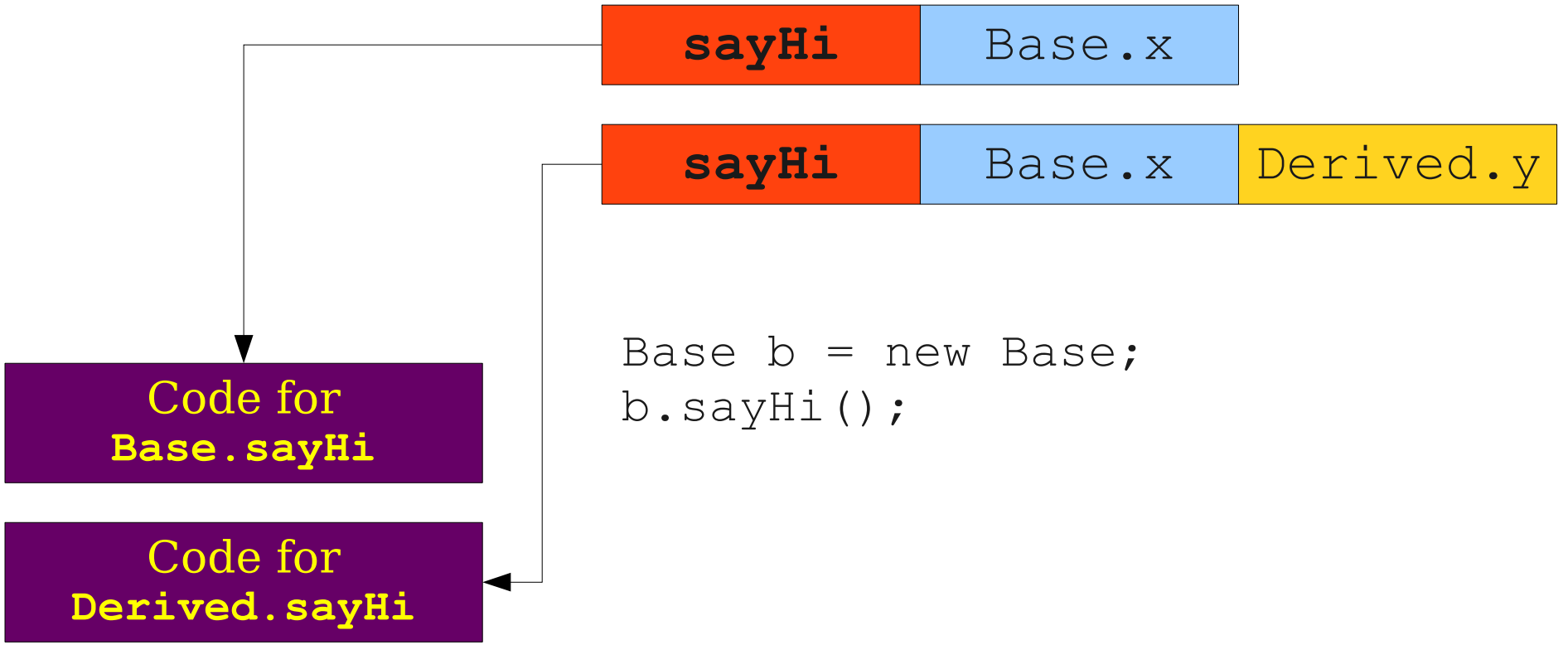
```
class Derived extends Base {
     int y;
     void sayHi() {
          Print("Derived");
     }
}
```



Virtual Function Tables

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

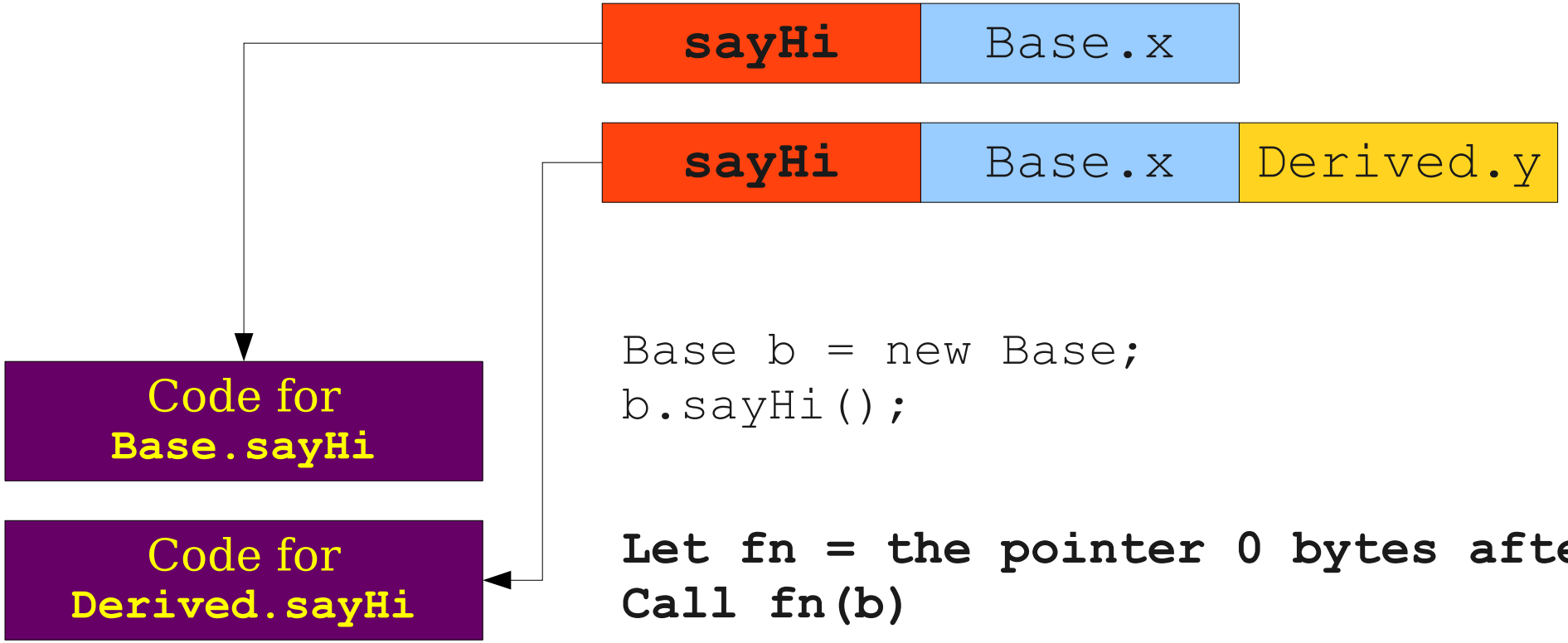
```
class Derived extends Base {
     int y;
     void sayHi() {
          Print("Derived");
     }
}
```



Virtual Function Tables

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

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



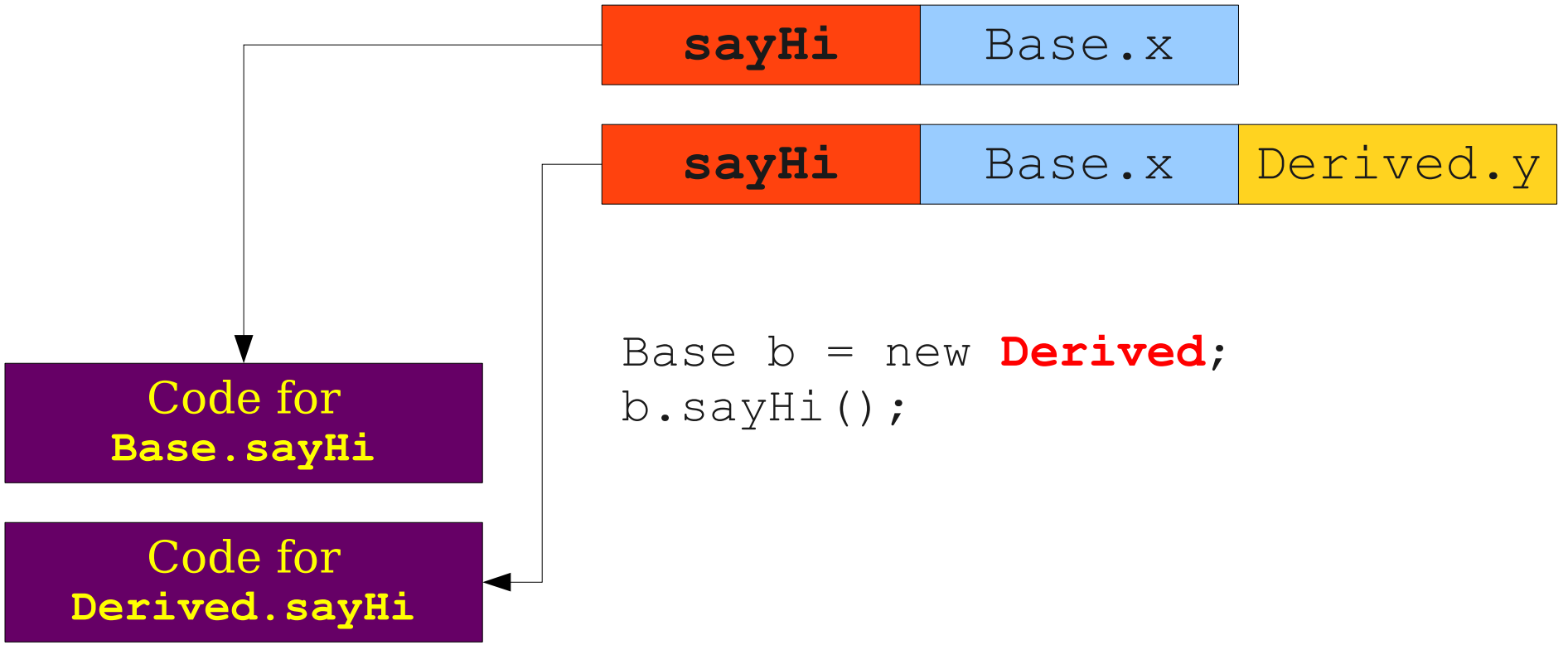

```
Base b = new Base;
b.sayHi();
```

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

Virtual Function Tables

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

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

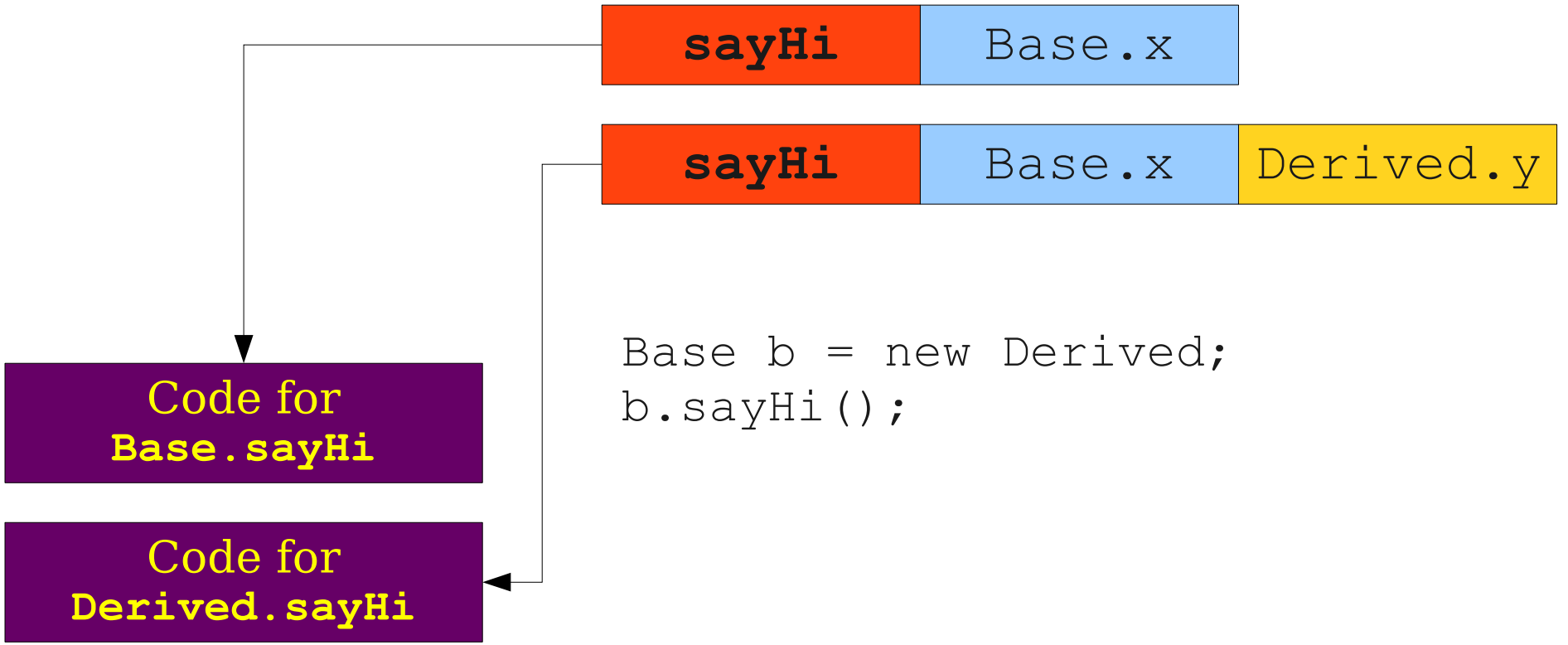



```
Base b = new Derived;
b.sayHi();
```

Virtual Function Tables

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

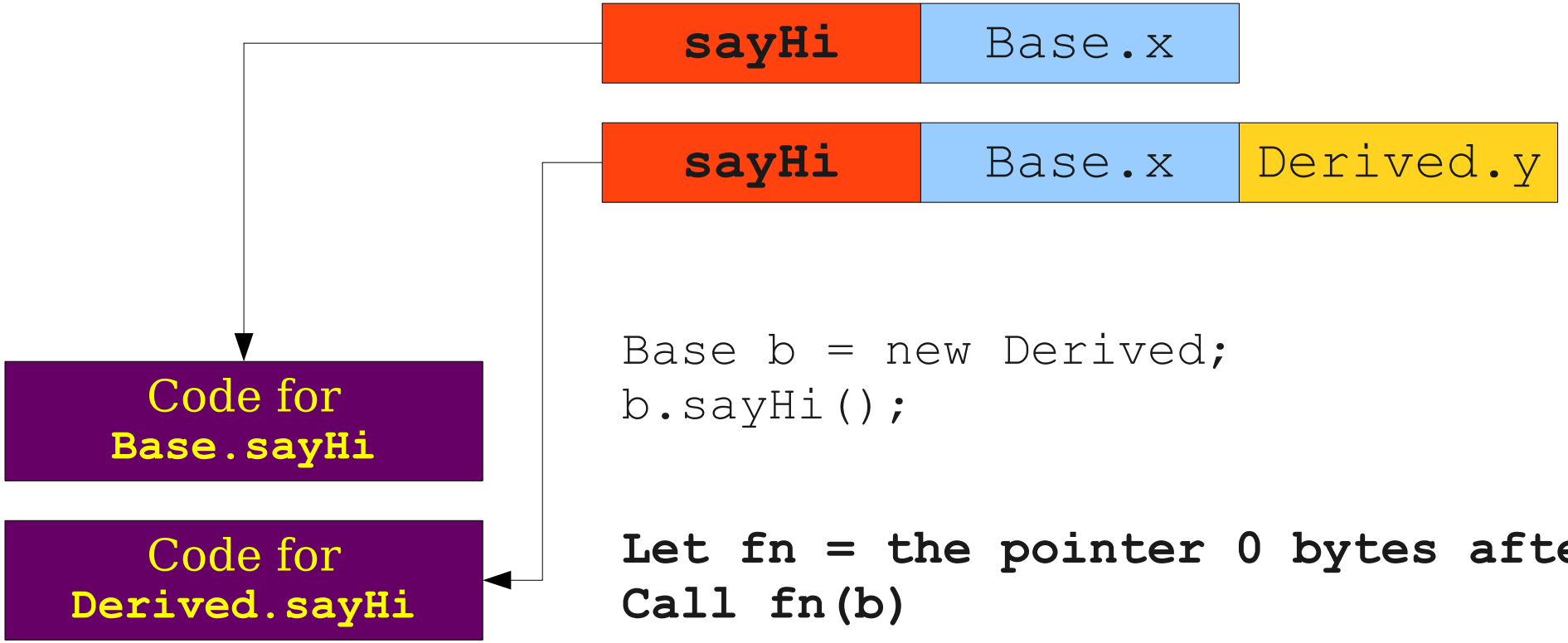
```
class Derived extends Base {
     int y;
     void sayHi() {
          Print("Derived");
     }
}
```



Virtual Function Tables

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

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




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

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

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;
    }
}
```

Code for **Base.sayHi**

Code for **Base.clone**

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;
    }
}
```

Code for **Base.sayHi**

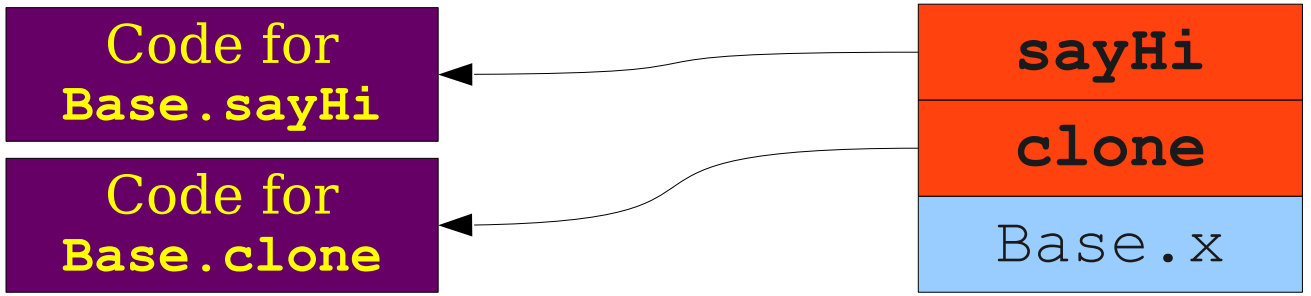
Code for **Base.clone**

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;
    }
}
```

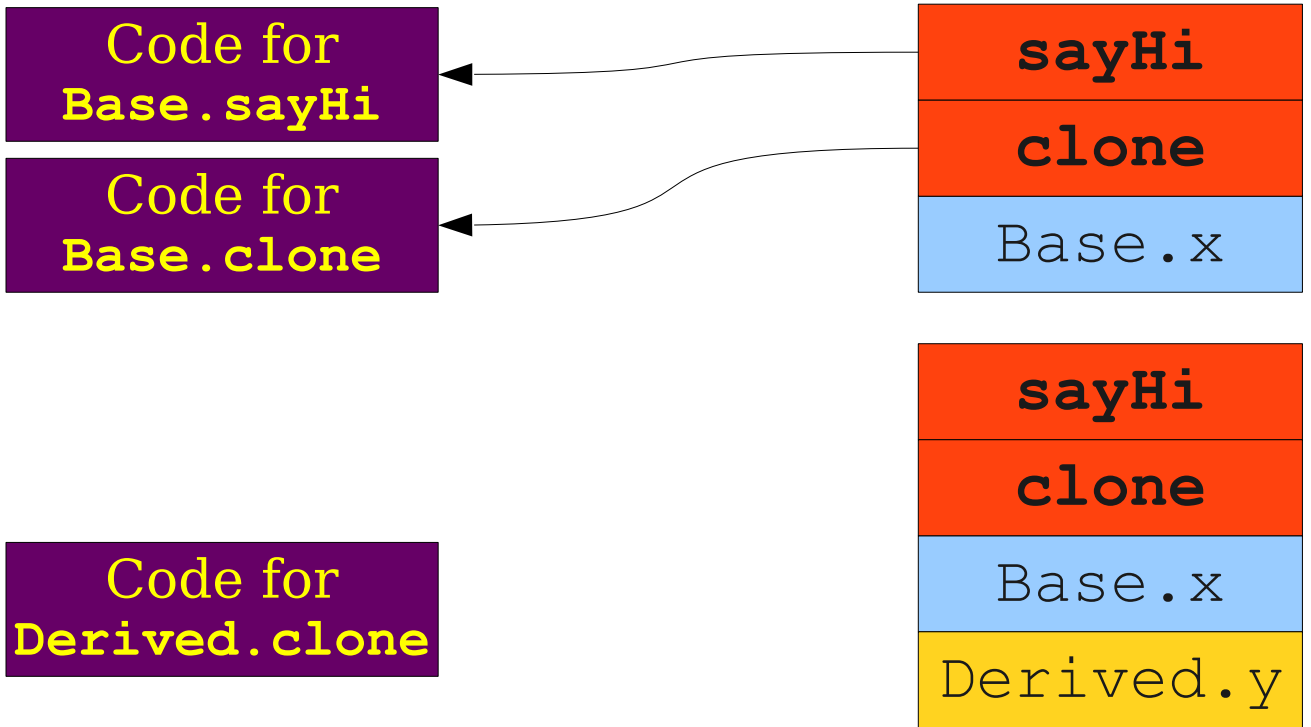


Code for
Derived.clone

More Virtual Function Tables

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

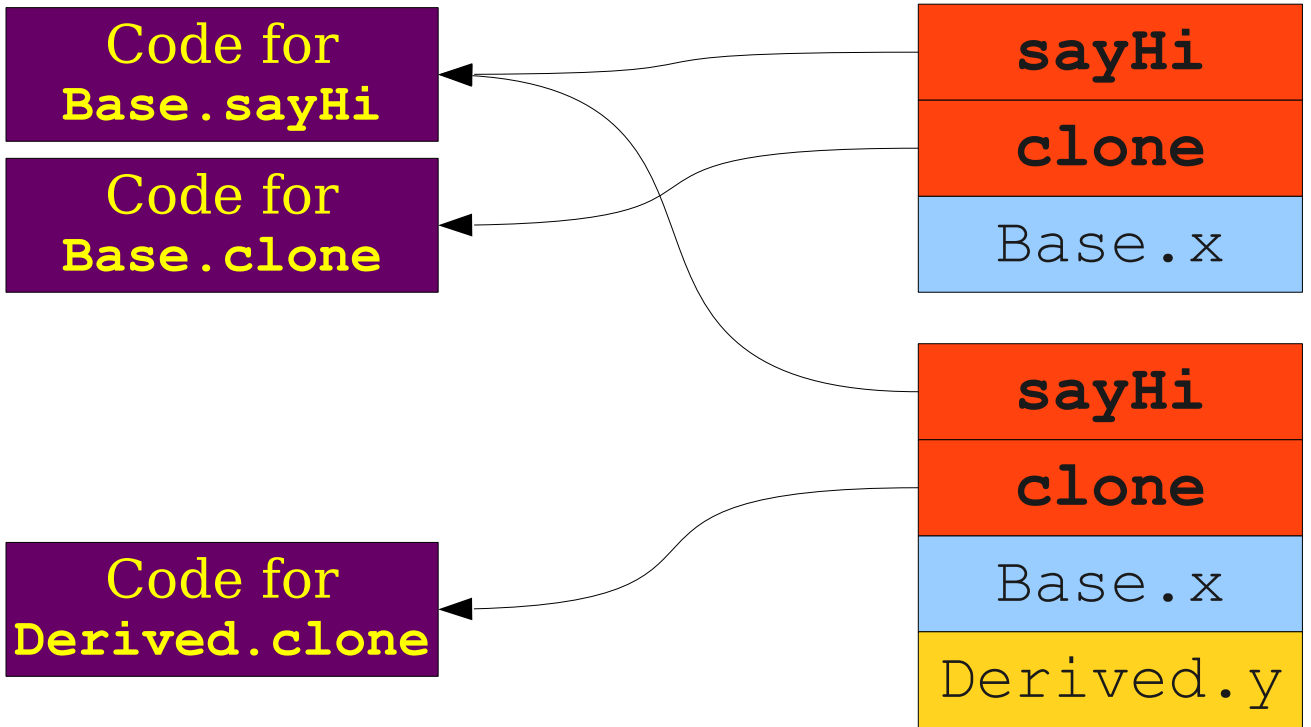
```
class Derived extends Base {
    int y;

    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?

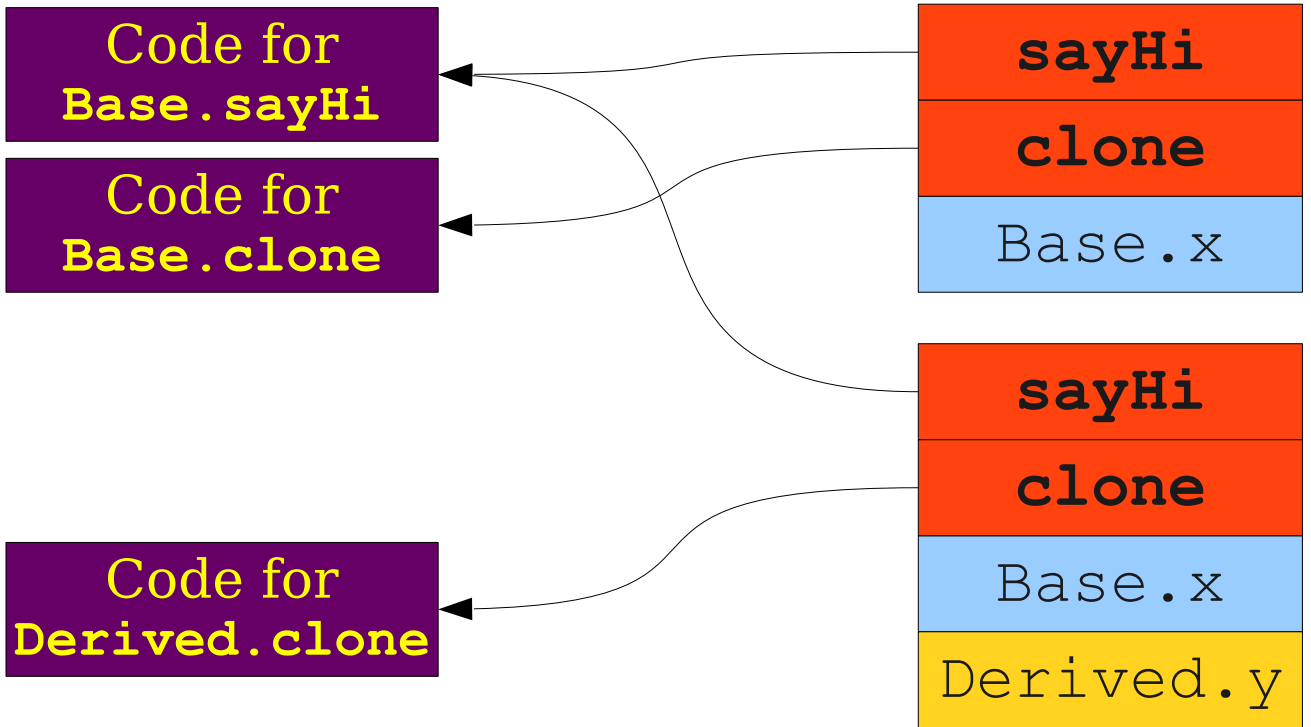
Analyzing our Approach

- 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;
    }
}
```

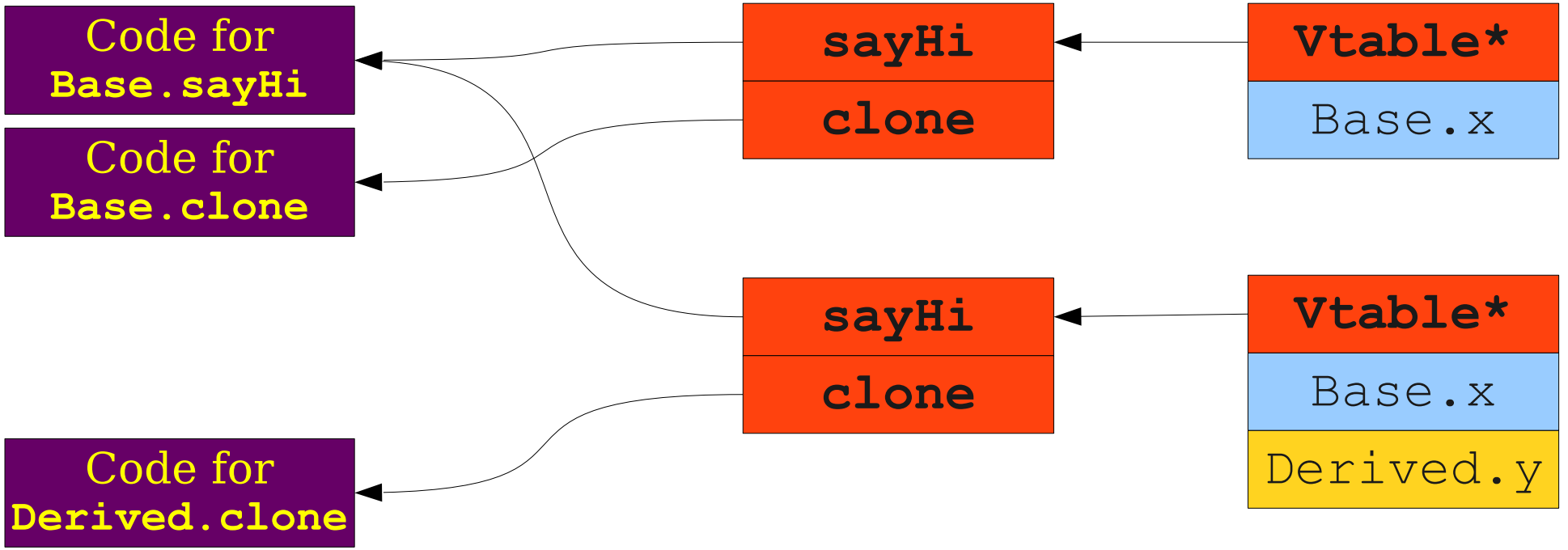
```
class Derived extends Base {
    int y;

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



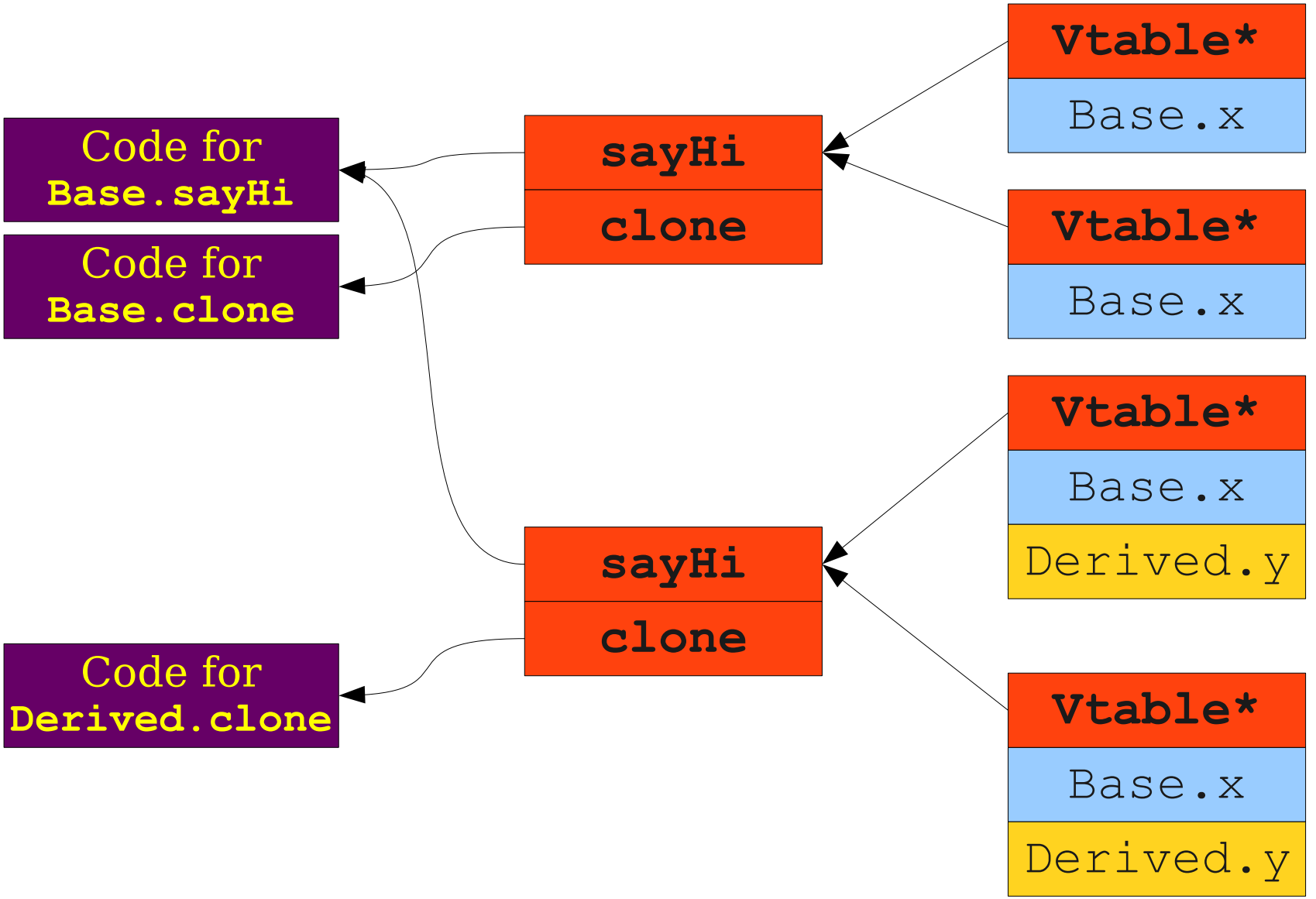
A Common Optimization

```
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();
```

```
> 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;
    }
}
```

```
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

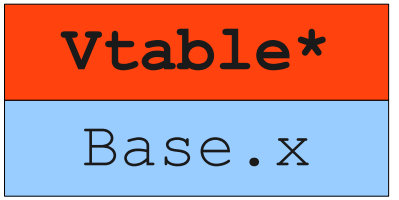
```
class Base {
    int x;
    void sayHi() {
        Print("Hi!");
    }
    Base clone() {
        return new Base;
    }
}
```

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

Code for **Base.sayHi**

Code for **Base.clone**

Code for **Derived.clone**



Inheritance without Vtables

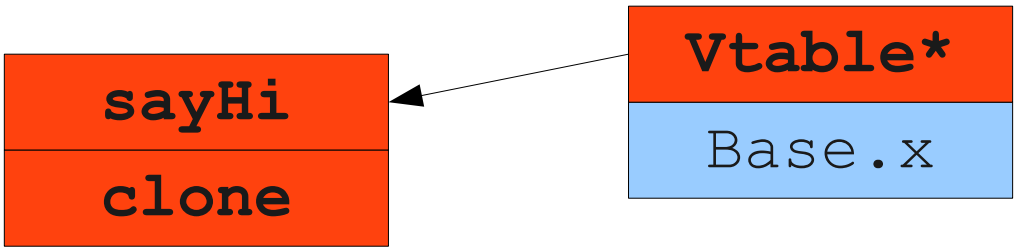
```
class Base {
    int x;
    void sayHi() {
        Print("Hi!");
    }
    Base clone() {
        return new Base;
    }
}
```

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

Code for **Base.sayHi**

Code for **Base.clone**

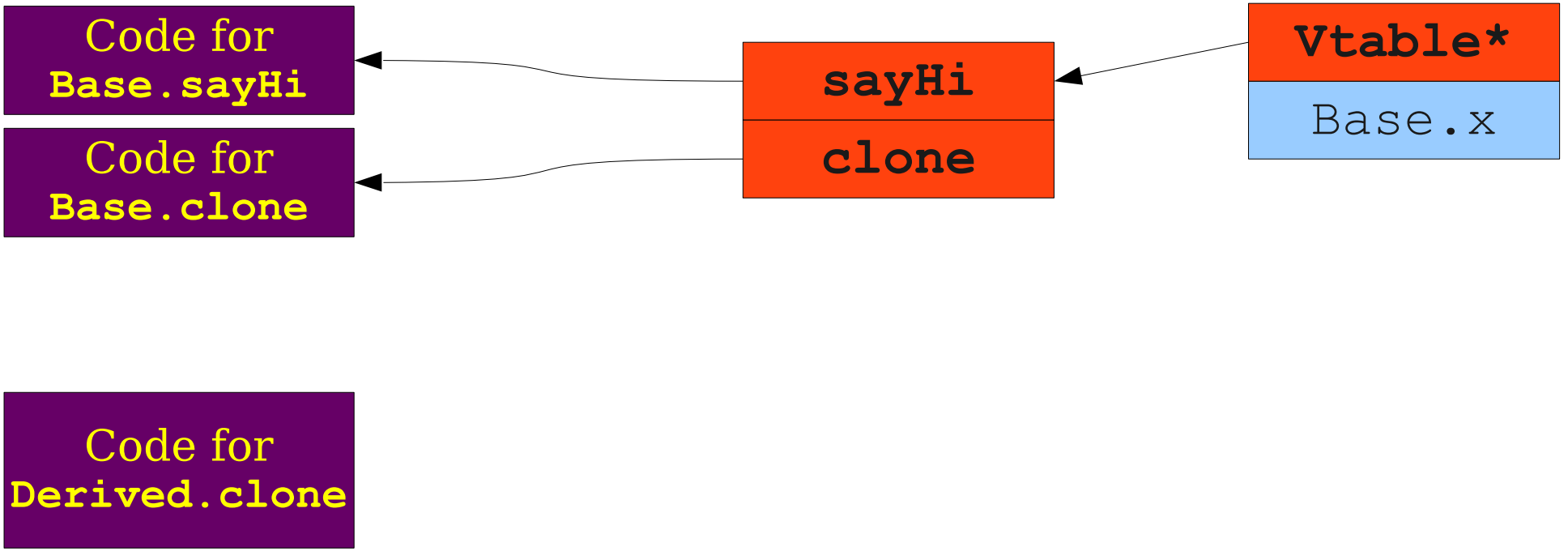
Code for **Derived.clone**



Inheritance without Vtables

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

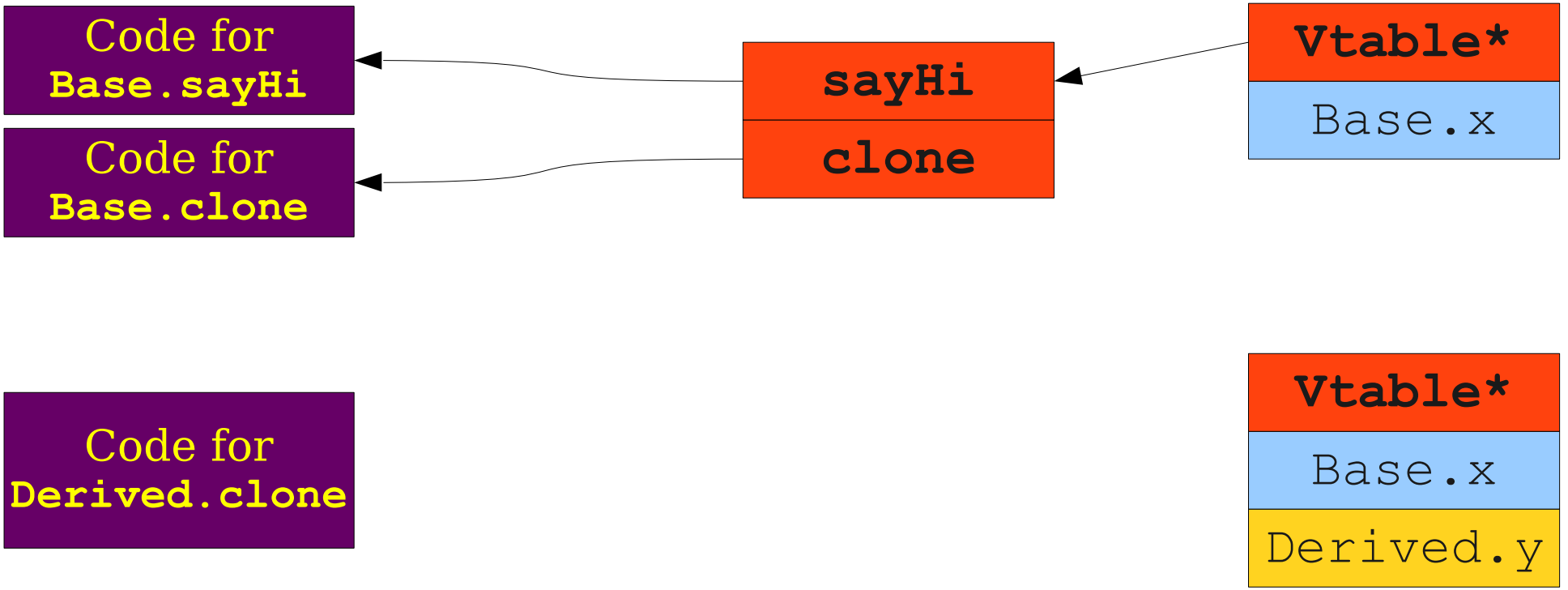
```
class Derived extends Base {
    int y;
    Derived clone() {
        return new Derived;
    }
}
```



Inheritance without Vtables

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

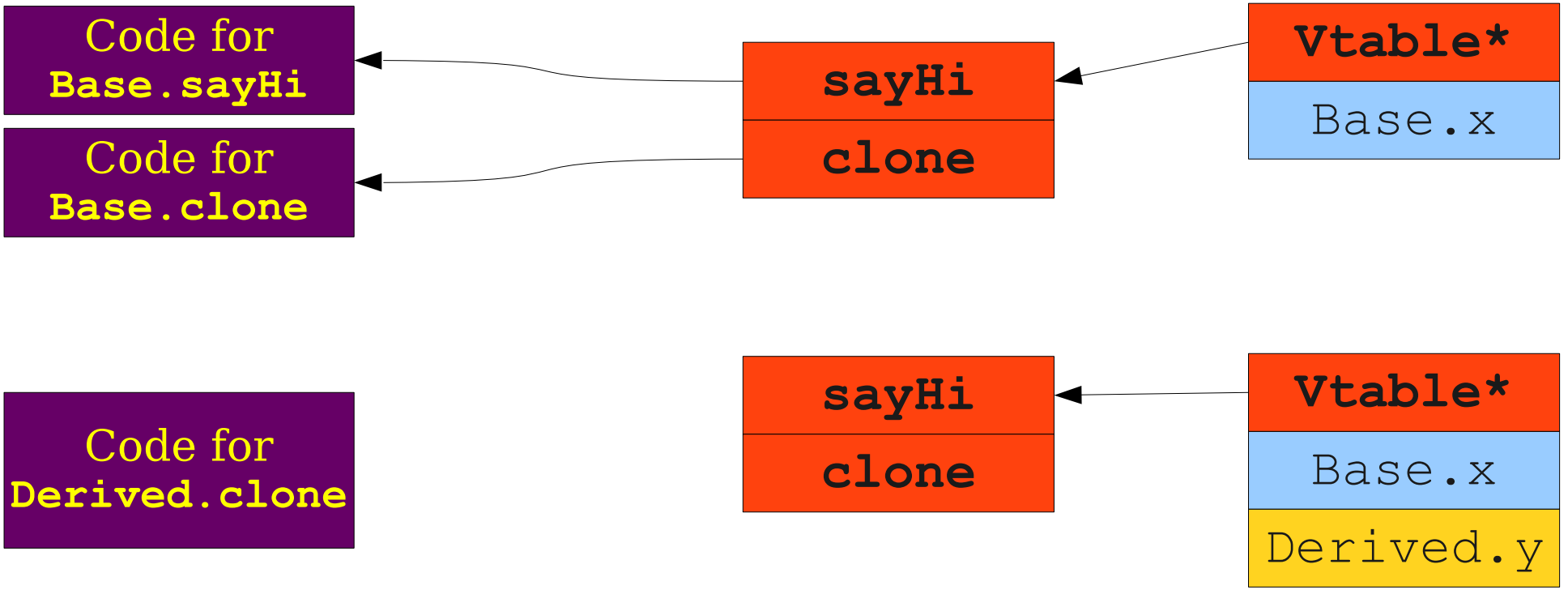
```
class Derived extends Base {
    int y;
    Derived clone() {
        return new Derived;
    }
}
```



Inheritance without Vtables

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

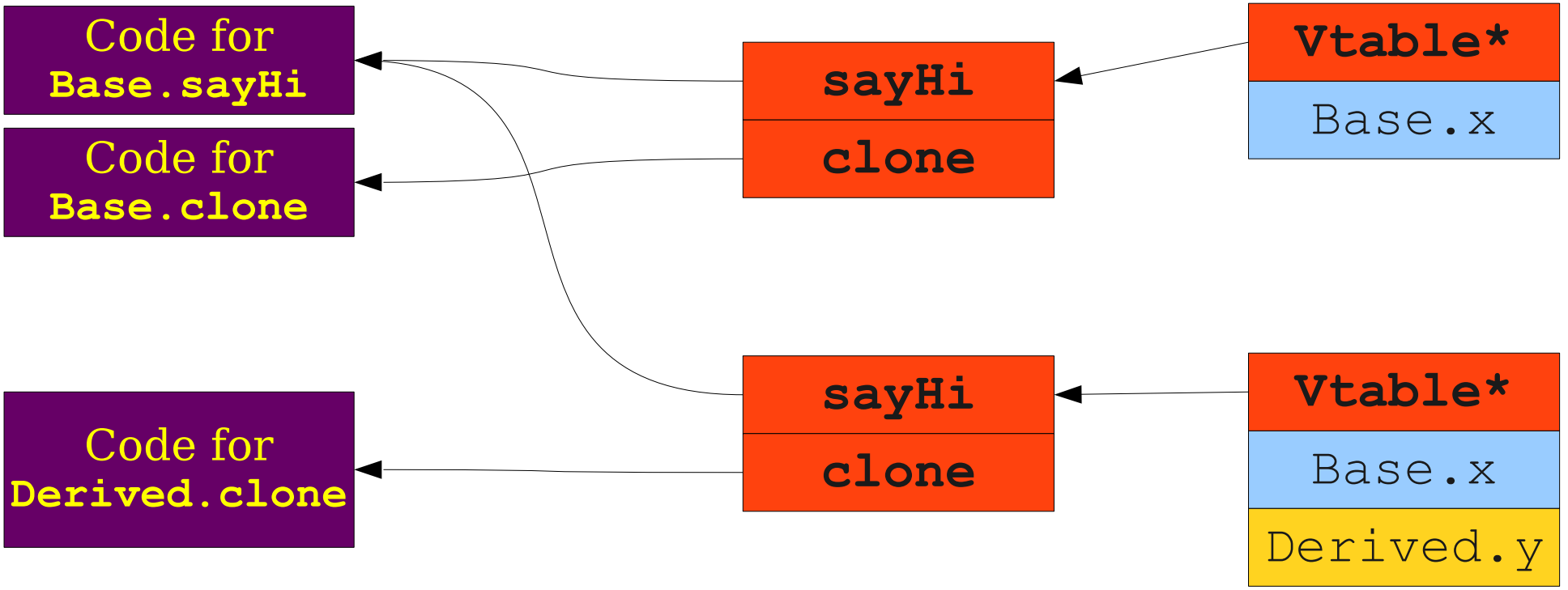
```
class Derived extends Base {
    int y;
    Derived clone() {
        return new Derived;
    }
}
```



Inheritance without Vtables

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

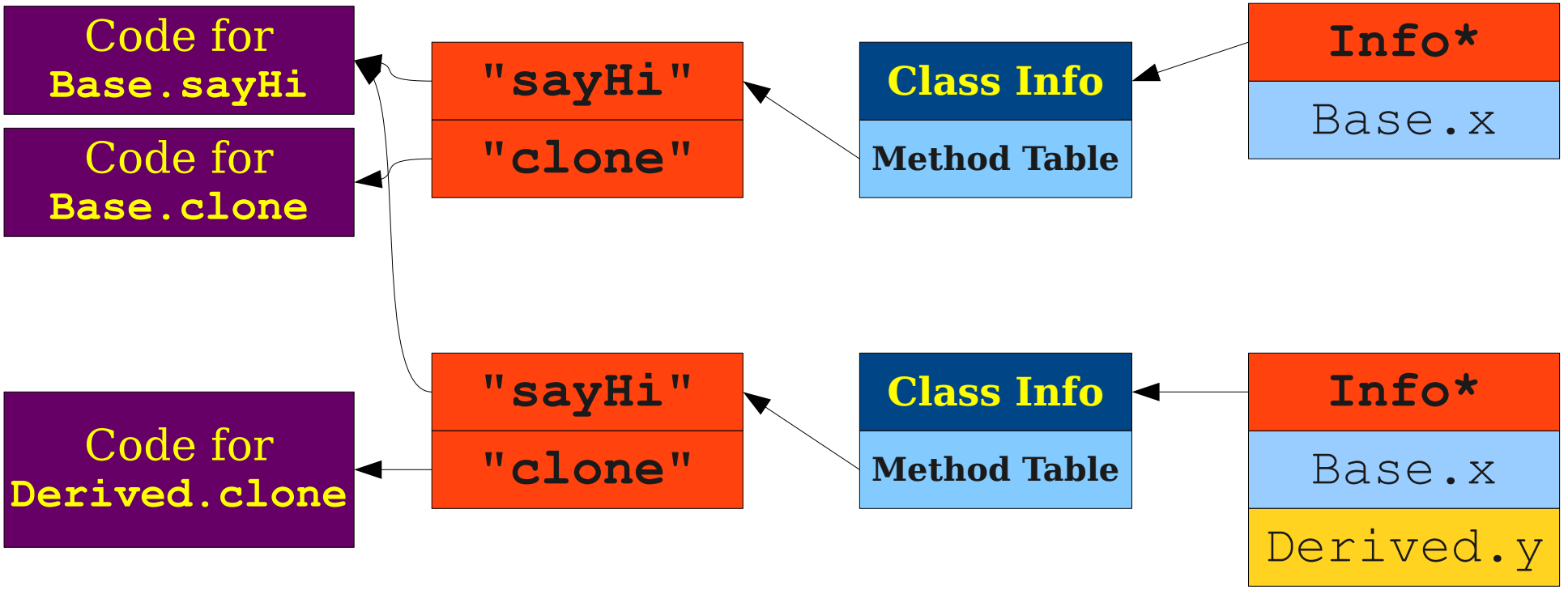
```
class Derived extends Base {
    int y;
    Derived clone() {
        return new Derived;
    }
}
```



Inheritance without Vtables

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

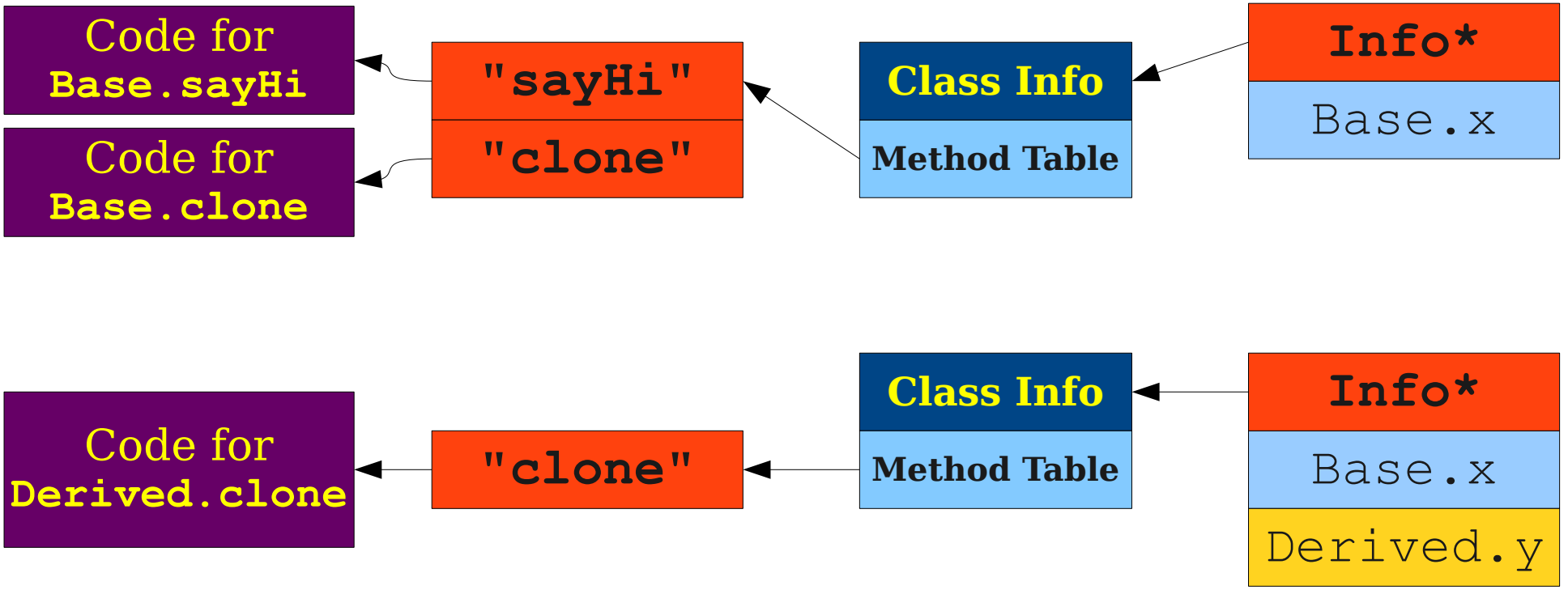
```
class Derived extends Base {
    int y;
    Derived clone() {
        return new Derived;
    }
}
```



Inheritance without Vtables

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

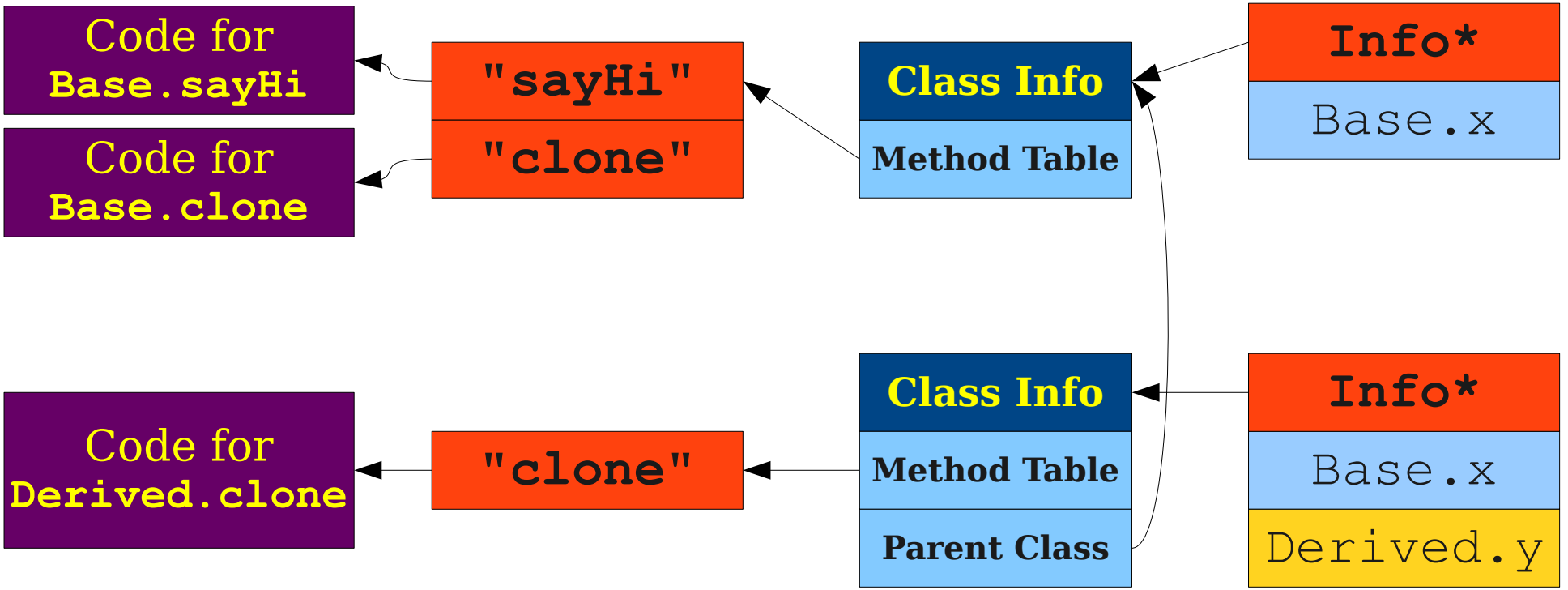
```
class Derived extends Base {
    int y;
    Derived clone() {
        return new Derived;
    }
}
```



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.

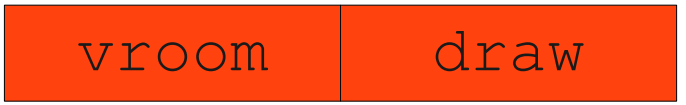
Vtables and Interfaces

```
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;
Visibie 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;
Visibie v2 = new Paint;
v1.draw();
v2.draw();
```

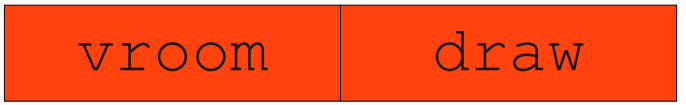
PaintedEngine vtable



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;
Visibie v2 = new Paint;
v1.draw();
v2.draw();
```

PaintedEngine vtable

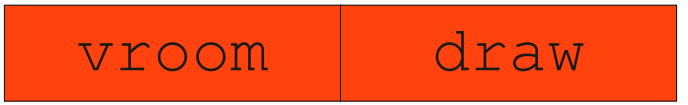


JetEngine vtable

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;
Visibie v2 = new Paint;
v1.draw();
v2.draw();
```

PaintedEngine vtable



JetEngine vtable

Paint vtable

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.
- We'll see two approaches for implementing interfaces efficiently.

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

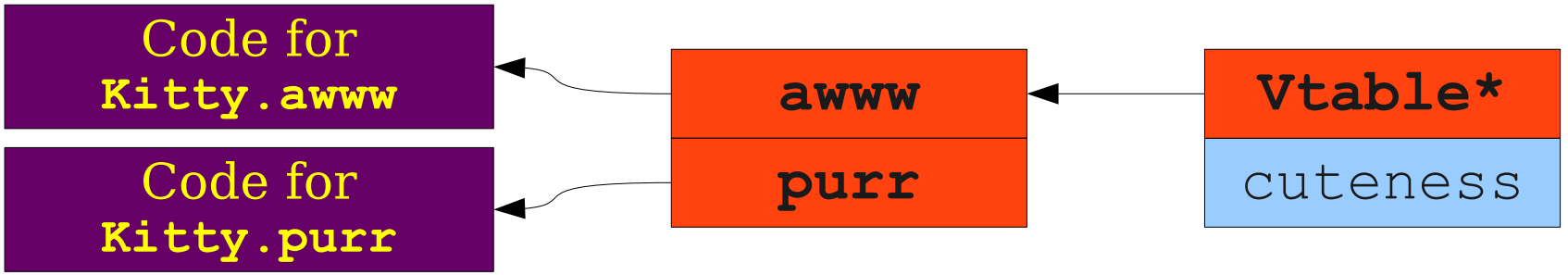
```
class Kitty implements Adorable {
    int cuteness;
    void awww() {
        Print("Meow");
    }
    void purr() {
        Print("Purr");
    }
}
```

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

Object Layout with Interfaces

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

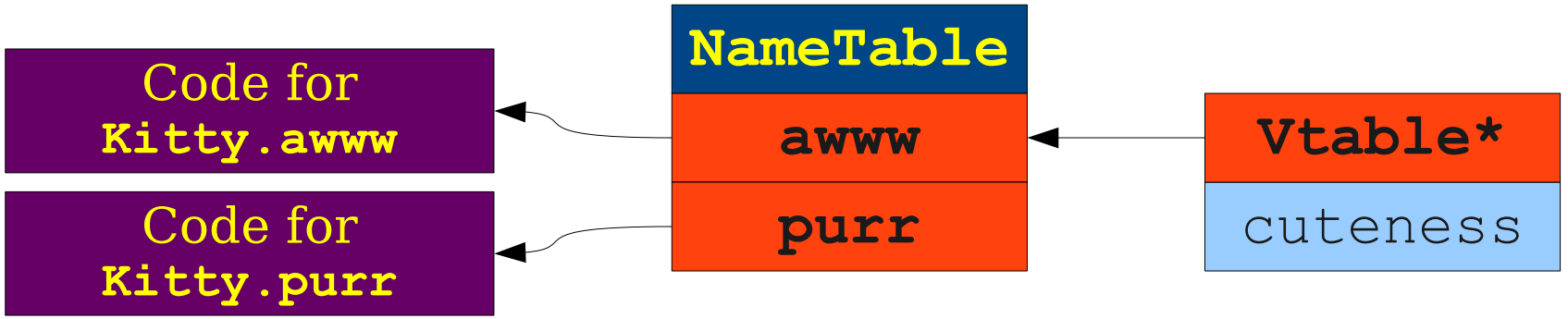
```
interface Adorable {
    void awww();
}
```



Object Layout with Interfaces

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

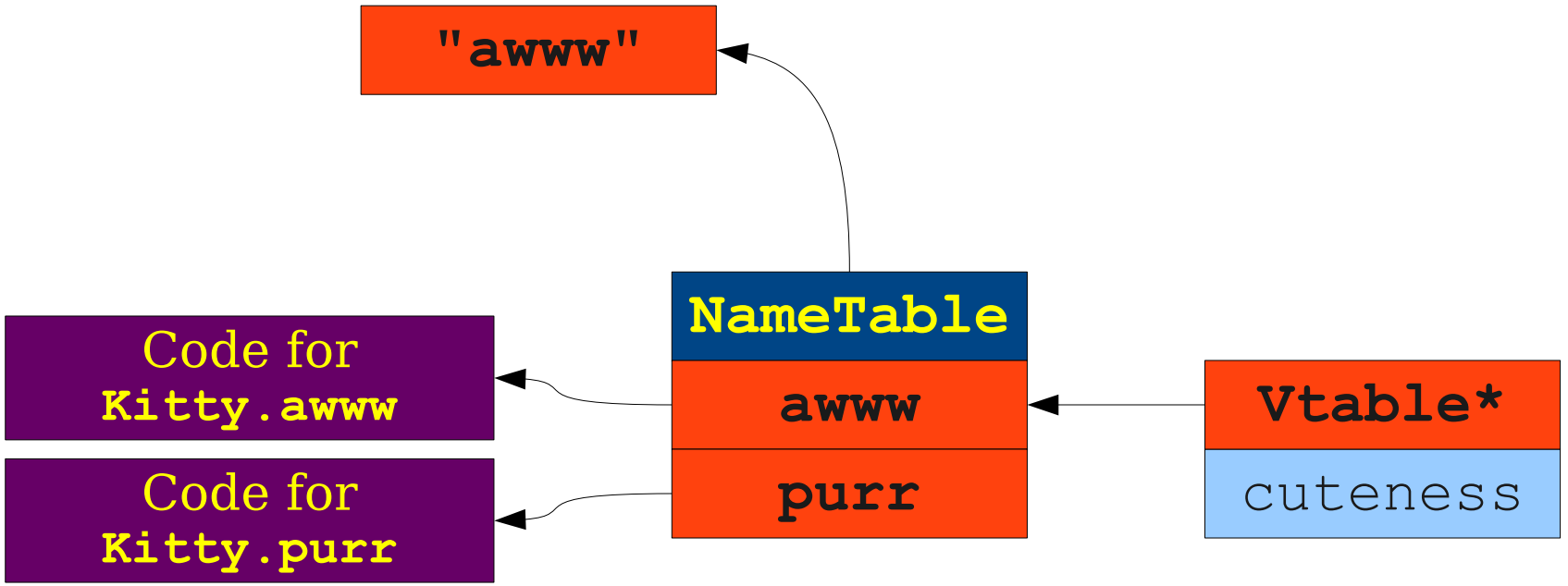
```
interface Adorable {
    void awww();
}
```



Object Layout with Interfaces

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

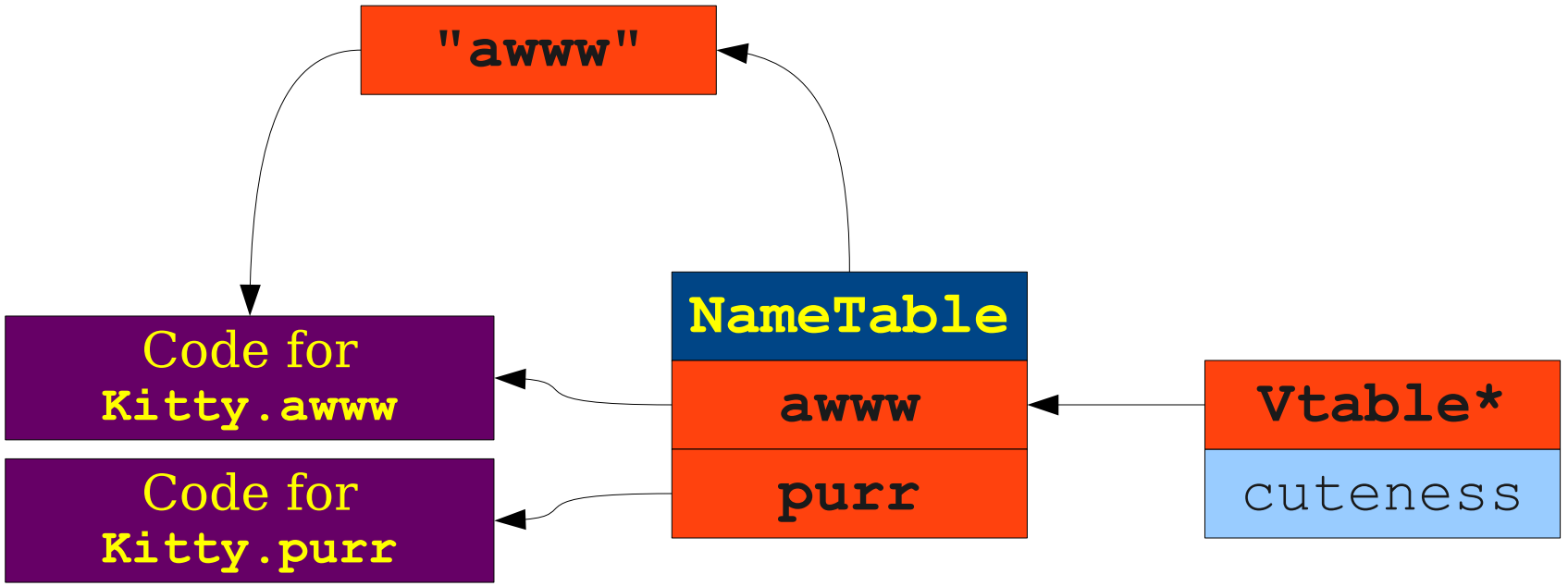
```
interface Adorable {
    void awww();
}
```



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.

Optimizing Interface Dispatch

- Assign a unique number to each interface method.
- Replace hash table of strings with hash table of integers.
  - Vtable effectively now a hash table instead of an array.
- Cost to do an interface dispatch now O(1).
  - (But still more expensive than a standard dynamic dispatch.)
- Would this work in PHP?

Optimizing Interface Dispatch

- Assign a unique number to each interface method.
- Replace hash table of strings with hash table of integers.
  - Vtable effectively now a hash table instead of an array.
- Cost to do an interface dispatch now O(1).
  - (But still more expensive than a standard dynamic dispatch.)
- Would this work in PHP?
  - **No**; can still do string-based lookups directly.

Optimizing Even Further

- Vtable lookups are (comparatively) fast compared to hash table lookups.
  - Can often do vtable lookup in two instructions!
- Hashing strings is (comparatively) very slow; hashing integers is (comparatively) slow.
- Can we eliminate the hash lookups in some cases?

An Observation

```
interface Kitty {
    void meow();
}
```

```
class NyanCat implements Kitty {
    void meow() {
        Print("Nyan!");
    }
}
```

```
class Garfield implements Kitty {
    void meow() {
        Print("I hate Mondays");
    }
}
```

```
Kitty n = new NyanCat;
for (i = 0; i < 100; ++i)
    n.meow();

Kitty g = new Garfield;
for (i = 0; i < 100; ++i)
    g.meow();
```

An Observation

```
interface Kitty {
    void meow();
}
```

```
class NyanCat implements Kitty {
    void meow() {
        Print("Nyan!");
    }
}
```

```
class Garfield implements Kitty {
    void meow() {
        Print("I hate Mondays");
    }
}
```

```
Kitty n = new NyanCat;
for (i = 0; i < 100; ++i)
    n.meow();

Kitty g = new Garfield;
for (i = 0; i < 100; ++i)
    g.meow();
```

Code for **NyanCat.meow**

Code for **Garfield.meow**

An Observation

```
interface Kitty {
    void meow();
}
```

```
class NyanCat implements Kitty {
    void meow() {
        Print("Nyan!");
    }
}
```

```
class Garfield implements Kitty {
    void meow() {
        Print("I hate Mondays");
    }
}
```

```
Kitty n = new NyanCat;
for (i = 0; i < 100; ++i)
    n.meow();

Kitty g = new Garfield;
for (i = 0; i < 100; ++i)
    g.meow();
```

Code for **NyanCat.meow**

Code for **Garfield.meow**

Remembering Dispatches

- A particular interface dispatch site often refers to the same method on the majority of its calls.
- **Idea**: Have a global variable for each call site that stores
  - The type of the last object used there, and
  - What method the call resolved to.
- When doing an interface dispatch, check whether the type of the receiver matches the cached type.
  - If so, just use the known method.
  - If not, fall back to the standard string-based dispatch.
- This is called **inline caching**.

Tradeoffs in Inline Caching

- For **monomorphic** call sites (only one object type actually getting used), inline caching can be a huge performance win.
- For **polymorphic** call sites (multiple object types getting used), inline caching can slow down the program.
  - (Why?)
- A more advanced technique called **polymorphic inline caching** tries to balance the two by maintaining a small collection of known types.
  - This optimization is used by many JITs for object-oriented languages, including Java.

Summary of String-Based Lookup

- Preserves the runtime speed of dispatch through objects.
- Allows flexible dispatch through interfaces.
- Can be optimized if method calls can be resolved statically.
- Can be optimized further using variants of inline caching.

Vtables Revisited

- **Recall**: Why do interfaces complicate vtable layouts?
- **Answer**: Interface methods must have a consistent position in all vtables.
- **Idea**: What if we have multiple vtables per object, one for each interface?
  - Allows interface methods to be positioned independently of one another.
  - Allows for fast vtable lookups relative to string-based approach.
- This is tricky but effective; most C++ implementations use this approach.

Interfaces and Vtables

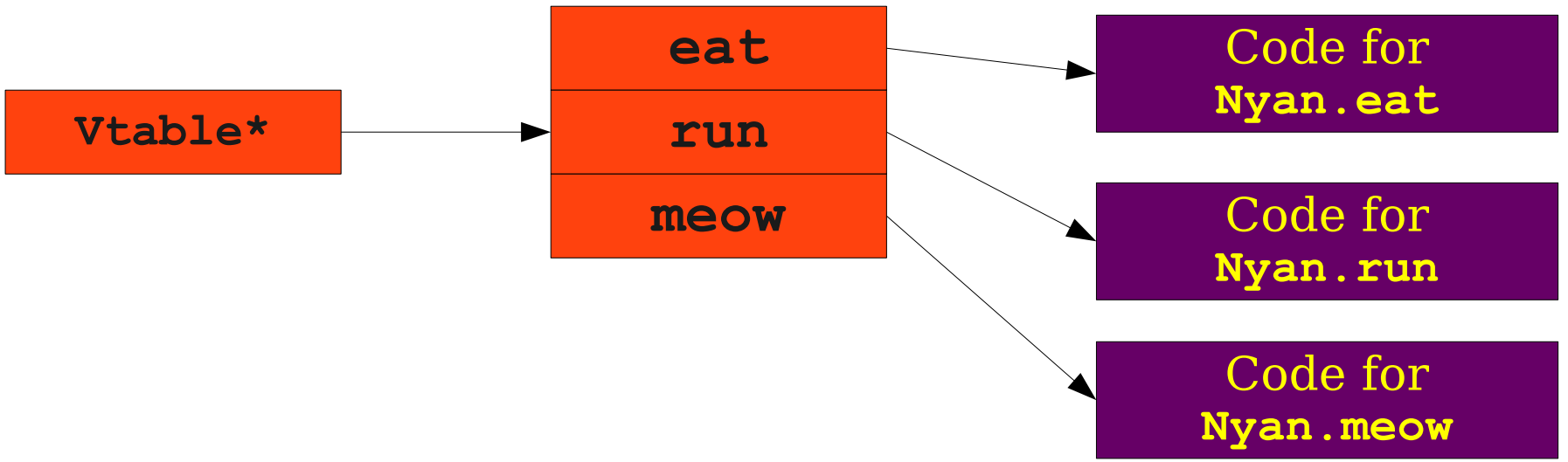
```
interface Meme {
    void run();
}
interface Cat {
    void meow();
}
```

```
class Nyan implements Meme, Cat {
    void run() {
        Print("Nyan!");
    }
    void meow() {
        Print("Nyan");
    }
    void eat() {
        Print("Yummy Pop Tart!");
    }
}
```

Interfaces and Vtables

```
interface Meme {
    void run();
}
interface Cat {
    void meow();
}
```

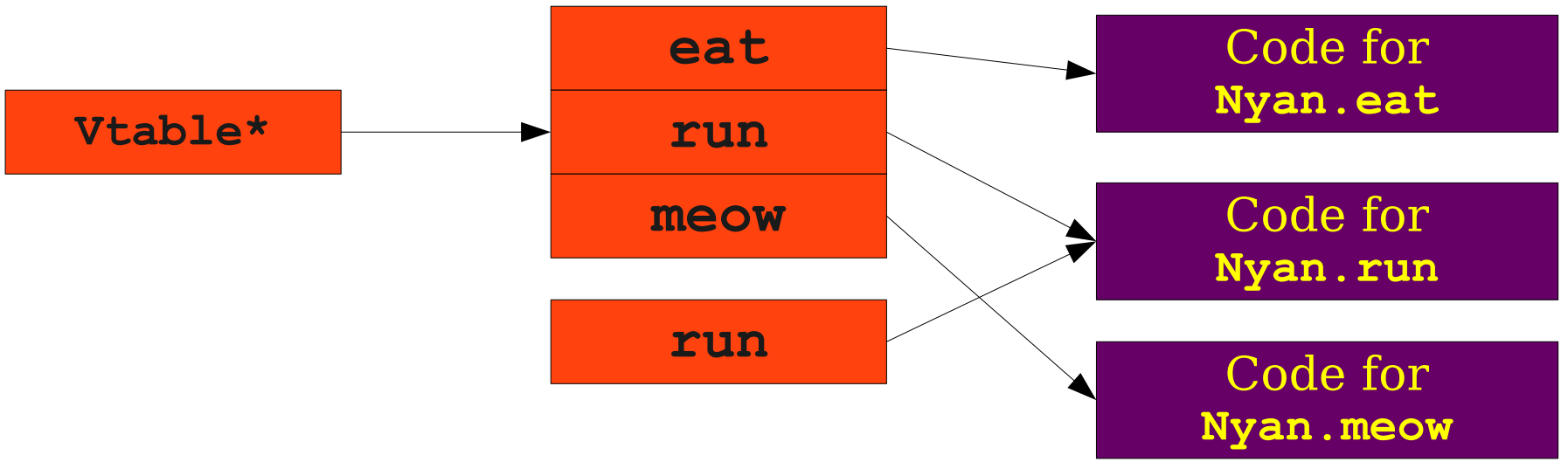
```
class Nyan implements Meme, Cat {
    void run() {
        Print("Nyan!");
    }
    void meow() {
        Print("Nyan");
    }
    void eat() {
        Print("Yummy Pop Tart!");
    }
}
```



Interfaces and Vtables

```
interface Meme {
    void run();
}
interface Cat {
    void meow();
}
```

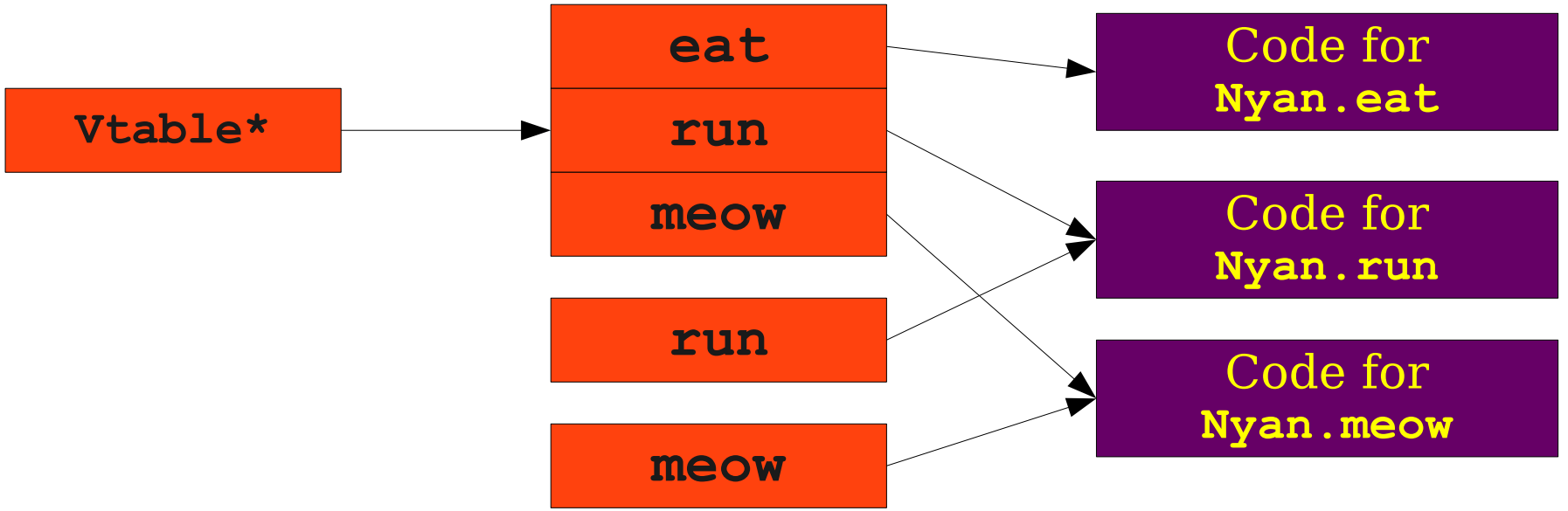
```
class Nyan implements Meme, Cat {
    void run() {
        Print("Nyan!");
    }
    void meow() {
        Print("Nyan");
    }
    void eat() {
        Print("Yummy Pop Tart!");
    }
}
```



Interfaces and Vtables

```
interface Meme {
    void run();
}
interface Cat {
    void meow();
}
```

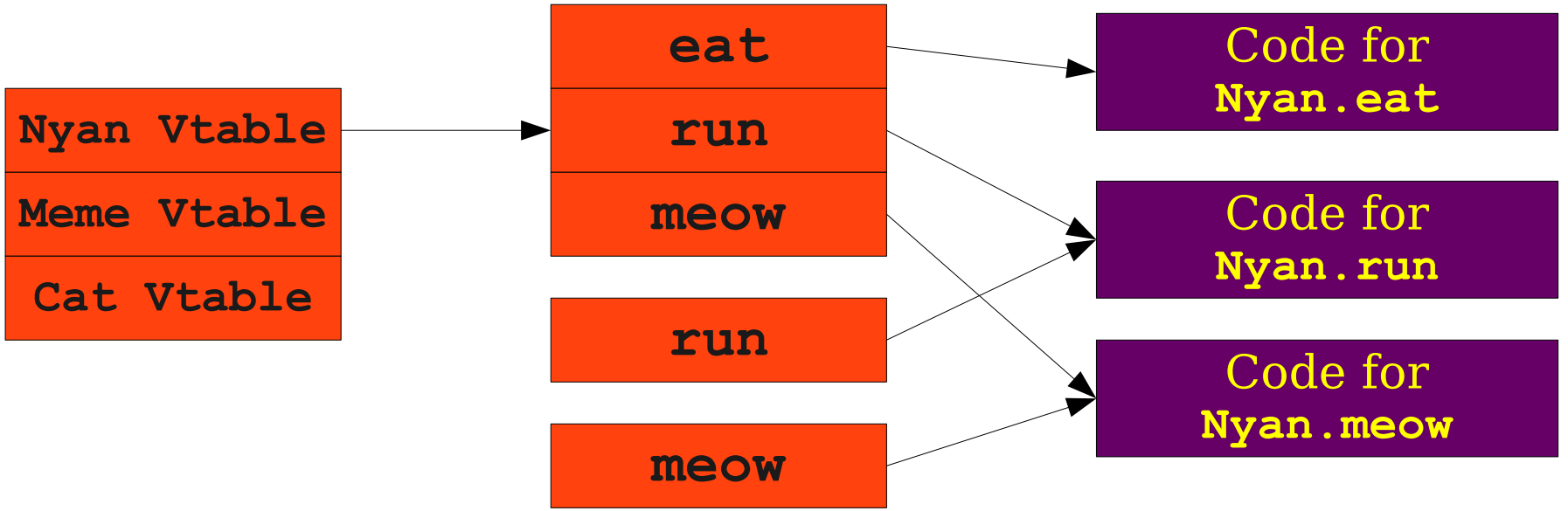
```
class Nyan implements Meme, Cat {
    void run() {
        Print("Nyan!");
    }
    void meow() {
        Print("Nyan");
    }
    void eat() {
        Print("Yummy Pop Tart!");
    }
}
```



Interfaces and Vtables

```
interface Meme {
    void run();
}
interface Cat {
    void meow();
}
```

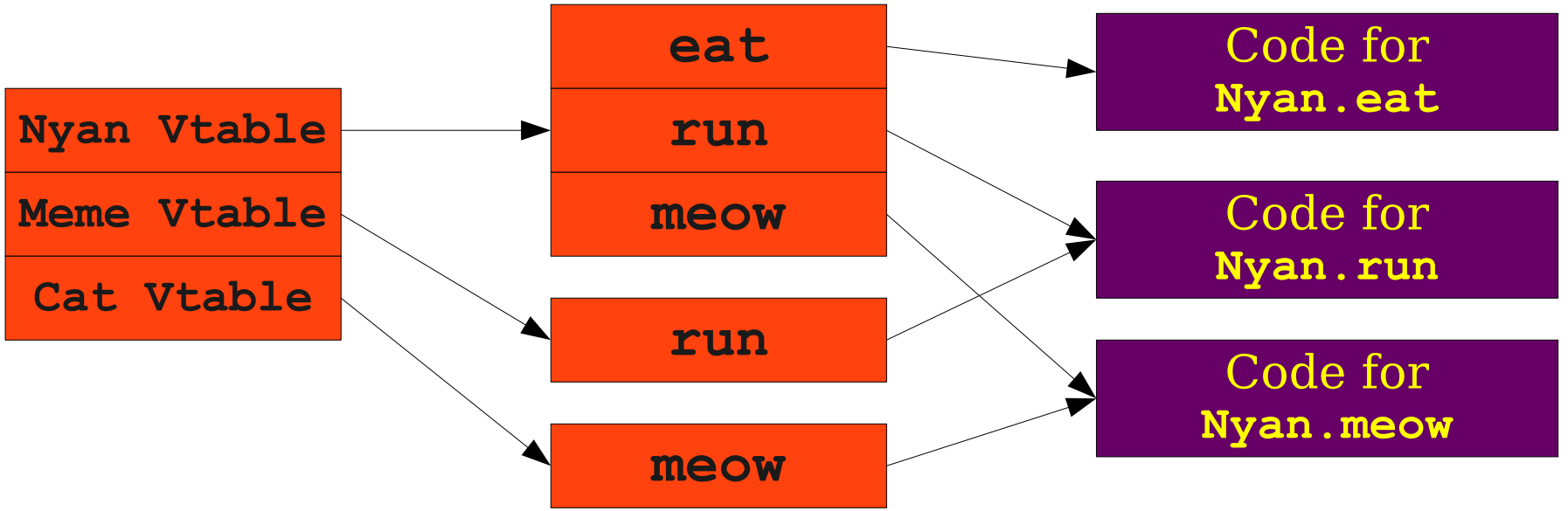
```
class Nyan implements Meme, Cat {
    void run() {
        Print("Nyan!");
    }
    void meow() {
        Print("Nyan");
    }
    void eat() {
        Print("Yummy Pop Tart!");
    }
}
```



Interfaces and Vtables

```
interface Meme {
    void run();
}
interface Cat {
    void meow();
}
```

```
class Nyan implements Meme, Cat {
    void run() {
        Print("Nyan!");
    }
    void meow() {
        Print("Nyan");
    }
    void eat() {
        Print("Yummy Pop Tart!");
    }
}
```



The Problem

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}


Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```

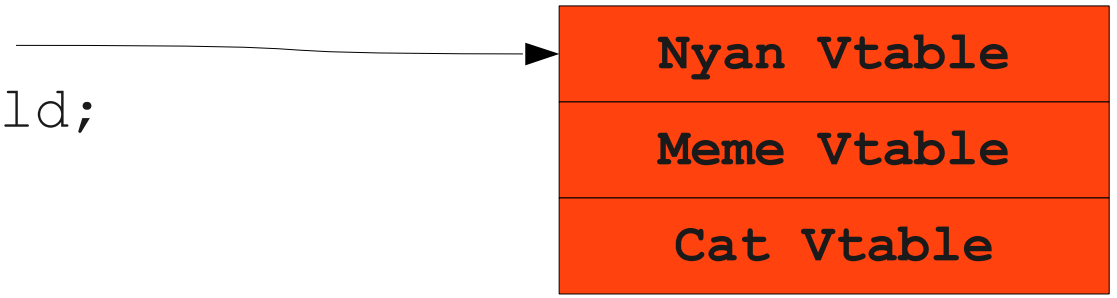
The Problem

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}


Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```

The Problem

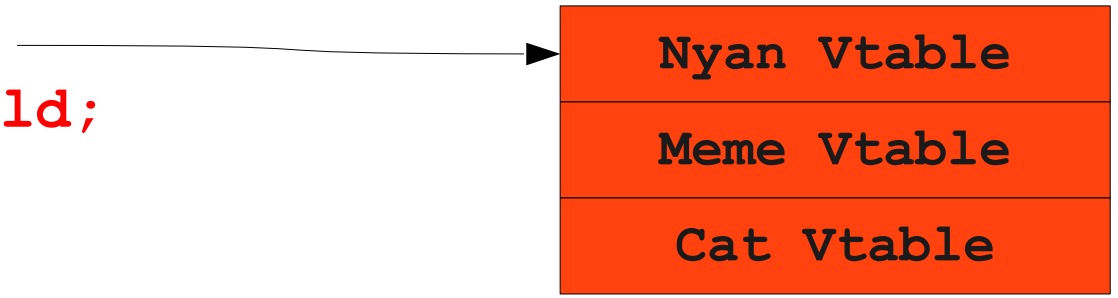
```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



The Problem

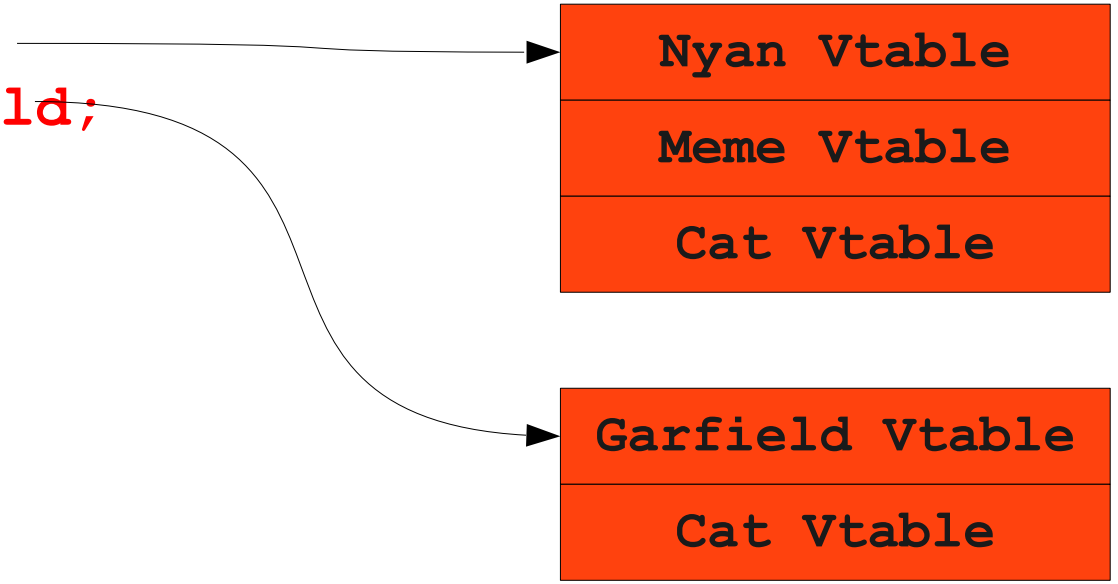
```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}


Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



The Problem

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

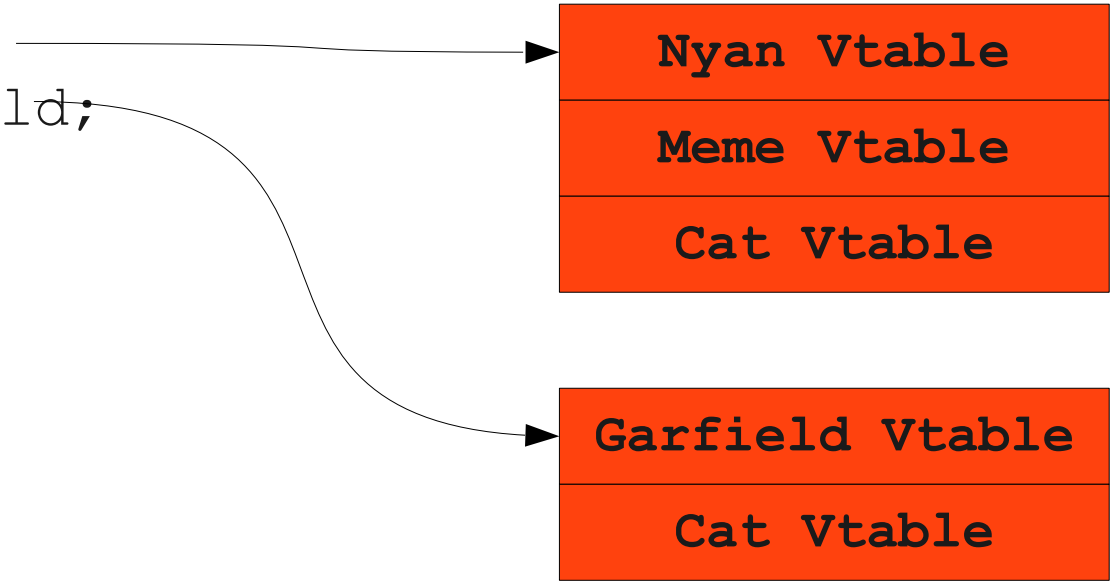
```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



The Problem

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

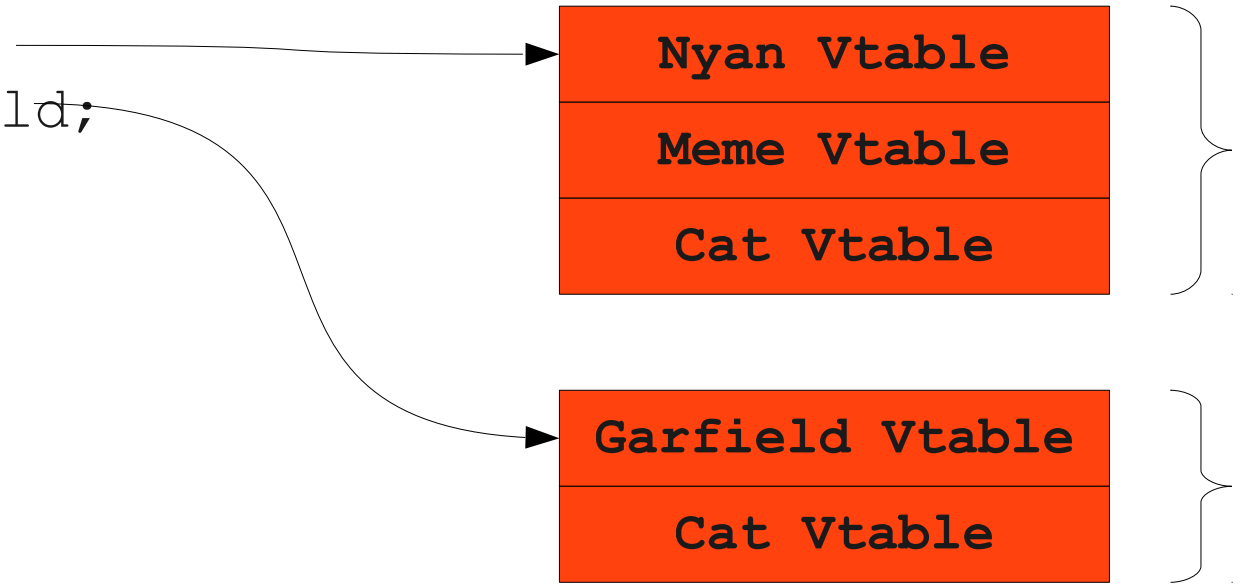
```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



The Problem

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



The Problem

- The offset from the base of the object to a particular interface vtable depends on the dynamic type of the object.
- We cannot generate IR code to do an interface dispatch without knowing where the vtable is.
- We don't seem to have gotten anywhere...

A Partial Solution

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}


Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```

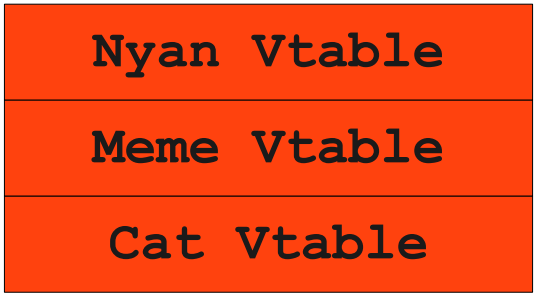
A Partial Solution

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}


Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```

A Partial Solution

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

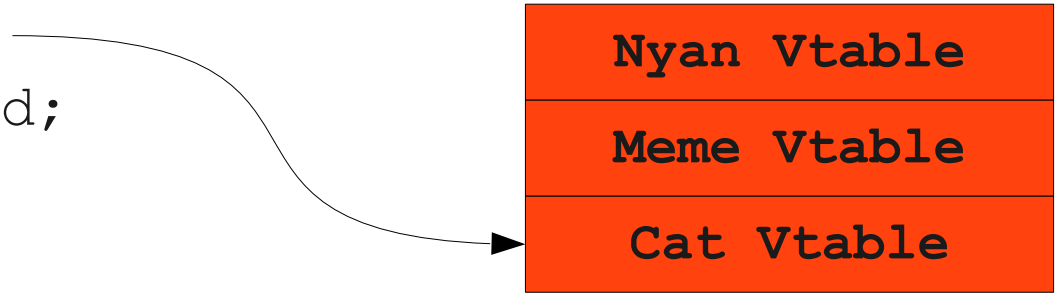
```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



A Partial Solution

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

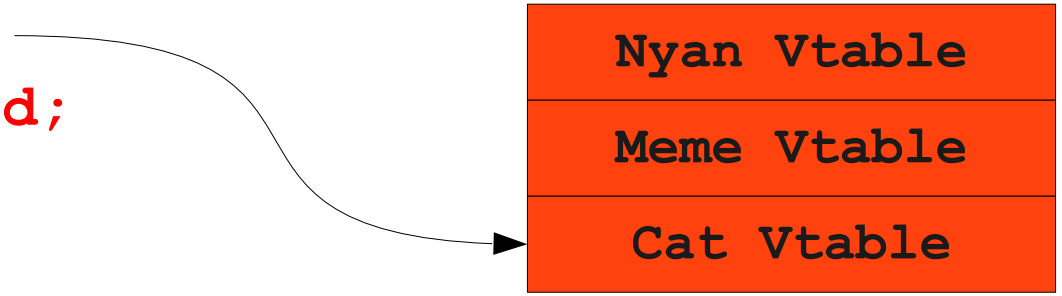
```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



A Partial Solution

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

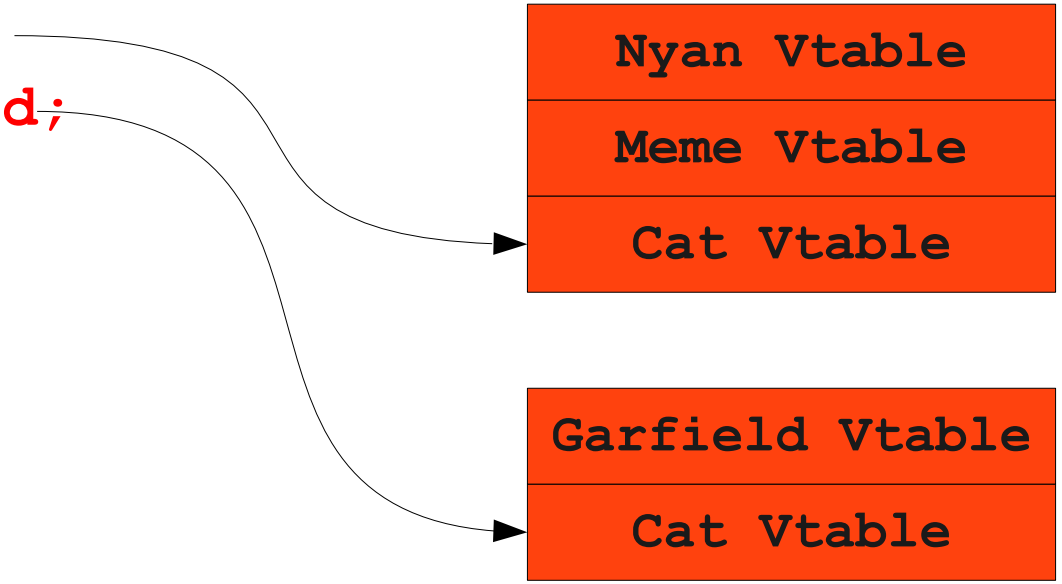
```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



A Partial Solution

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

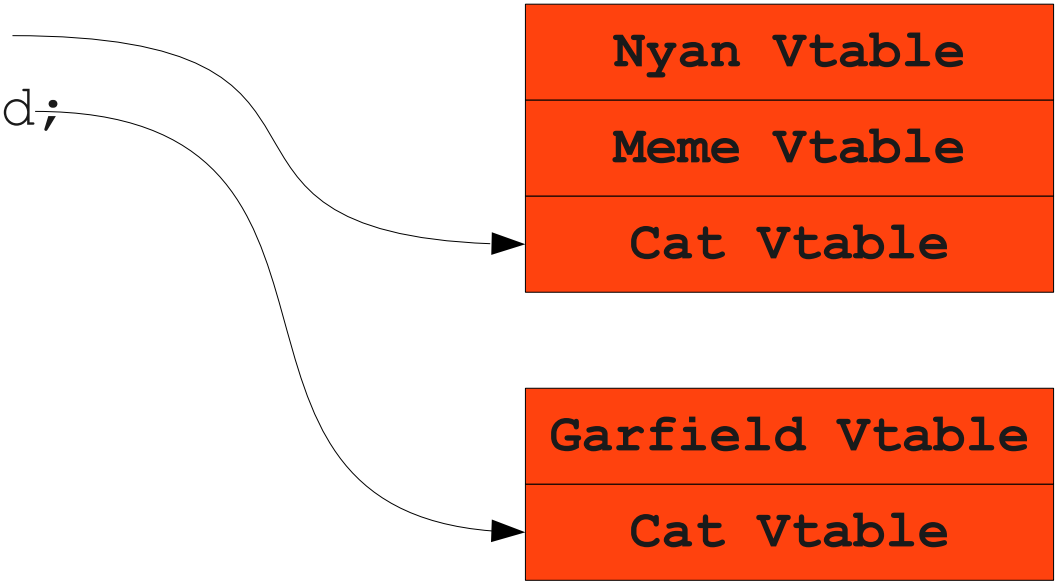
```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



A Partial Solution

```
class Nyan implements Meme, Cat {
    /* … */
}

class Garfield implements Cat {
    /* … */
}
```

```
Cat c1 = new Nyan;
Cat c2 = new Garfield;

c1.meow();
c2.meow();
```



A Partial Solution

- When upcasting an object to an interface type, change where the pointer points so that it sees the vtable pointer for that interface.
- We can now assume an interface reference refers directly to the vtable.
- But there's a ***serious*** problem with this implementation...

Looking in the Wrong Place

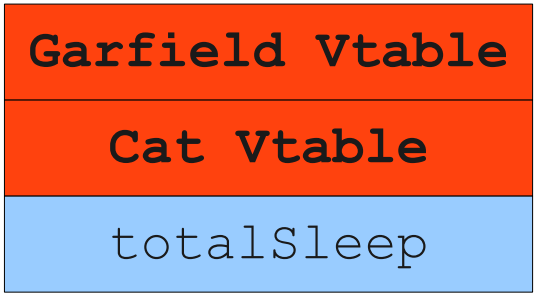
```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}

Cat g = new Garfield;
g.meow();
```

Looking in the Wrong Place

```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}

Cat g = new Garfield;
g.meow();
```

Looking in the Wrong Place

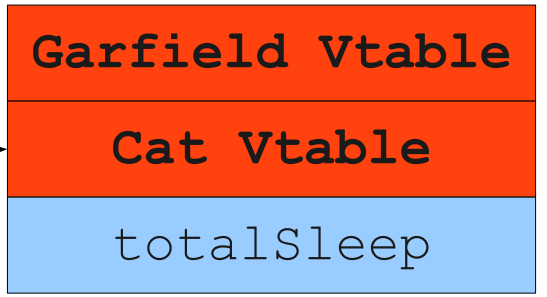
```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}
```




```
Cat g = new Garfield;
g.meow();
```

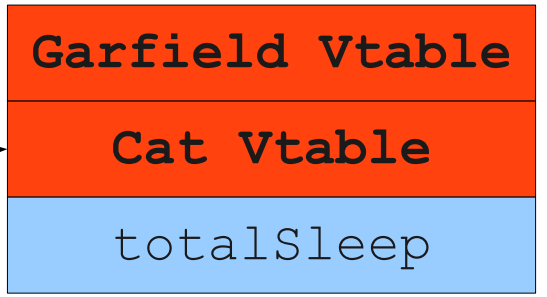
Looking in the Wrong Place

```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}

Cat g = new Garfield;
g.meow();
```



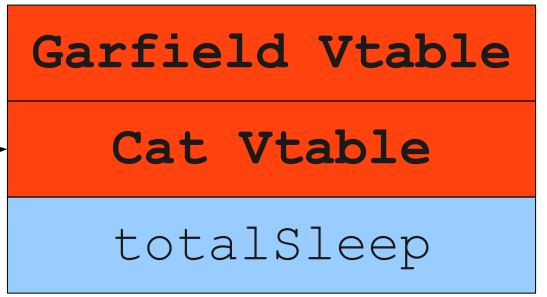
Looking in the Wrong Place

```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}

Cat g = new Garfield;
g.meow();
```



Looking in the Wrong Place

```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}

Cat g = new Garfield;
g.meow();
```

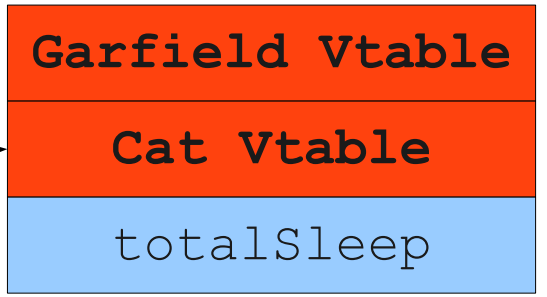


Code for Garfield::meow(Garfield* this)

Look up the integer 8 bytes past 'this'
Read its value into memory
Subtract one from the value
Store the value back into memory

Looking in the Wrong Place

```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}

Cat g = new Garfield;
g.meow();
```

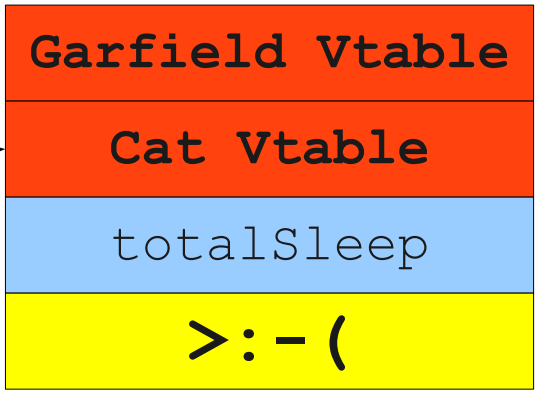


Code for Garfield::meow(Garfield* this)

Look up the integer 8 bytes past 'this'
Read its value into memory
Subtract one from the value
Store the value back into memory

Looking in the Wrong Place

```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm tired.");
    }
}

Cat g = new Garfield;
g.meow();
```



Code for Garfield::meow(Garfield* this)

- **Look up the integer 8 bytes past 'this'**
- **Read its value into memory**
- **Subtract one from the value**
- **Store the value back into memory**

Looking in the Wrong Place

- Interface pointers cannot be used directly as the `this` pointer in methods calls.
  - Pointing into the **middle** of an object, not the **base** of the object.
  - All field offsets will refer to the wrong parts of memory.
- How can we correct this?

Adding in Deltas

```
Cat g = new Garfield;
g.meow();
```

```
Code for Garfield::meow(Garfield* this)
    Look up the integer 8 bytes past 'this'
    Read its value into memory
    Subtract one from the value
    Store the value back into memory
```

Adding in Deltas

```
Cat g = new Garfield;
g.meow();
```

```
Code for Garfield::meow(Garfield* this)
    Look up the integer 8 bytes past 'this'
    Read its value into memory
    Subtract one from the value
    Store the value back into memory
```

Vtable Deltas

- Augment each interface vtable with the offset in bytes the pointer must be corrected to get back to the base of the object.
- A dynamic dispatch then looks like this:
  - Look up the address of the function to call by following the vtable pointer and looking at the recovered address.
  - Look up the amount to adjust the object pointer in the vtable.
  - Update the object pointer by adding in the given delta.
  - Call the function indicated in the vtable.

Analysis of Vtable Deltas

- Cost to invoke a method is O(1) regardless of the number of interfaces.
- Also a **fast** O(1); typically much better than a hash table lookup.
- Size of an object increases by O(`I`), where `I` is the number of interfaces.
- Cost to create an object is O(`I`), where `I` is the number of interfaces.
  - (Why?)

Comparison of Approaches

- String-based lookups have small objects and fast object creation but slow dispatch times.
  - Only need to set one vtable pointer in the generated object.
  - Dispatches require some type of string comparisons.
- Vtable-based lookups have larger objects and slower object creation but faster dispatch times.
  - Need to set multiple vtable pointers in the generated object.
  - Dispatches can be done using simple arithmetic.

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() {}
}

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.f

B.f

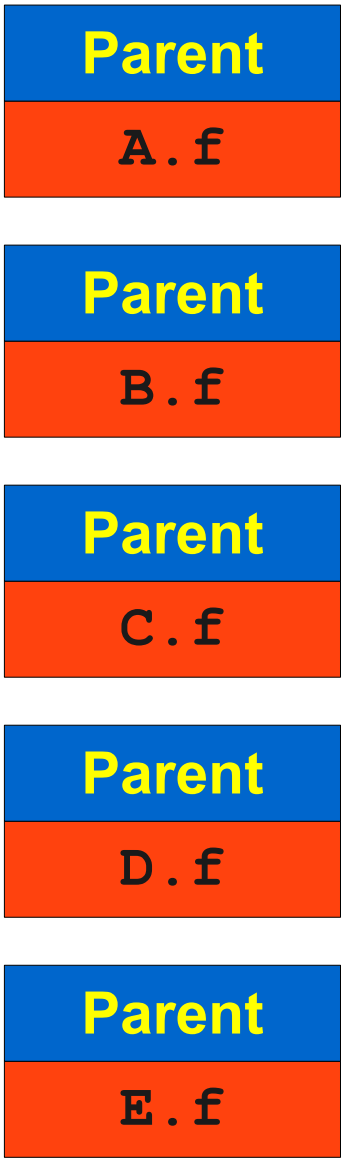
C.f

D.f

E.f

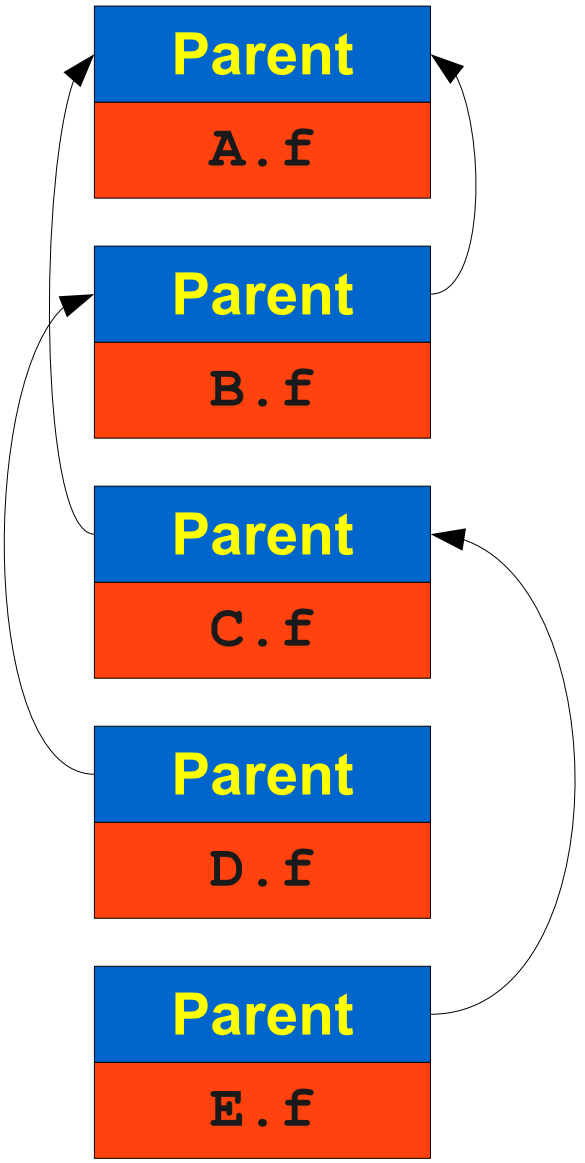
A Pretty Good Approach

```
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() {}
}
```



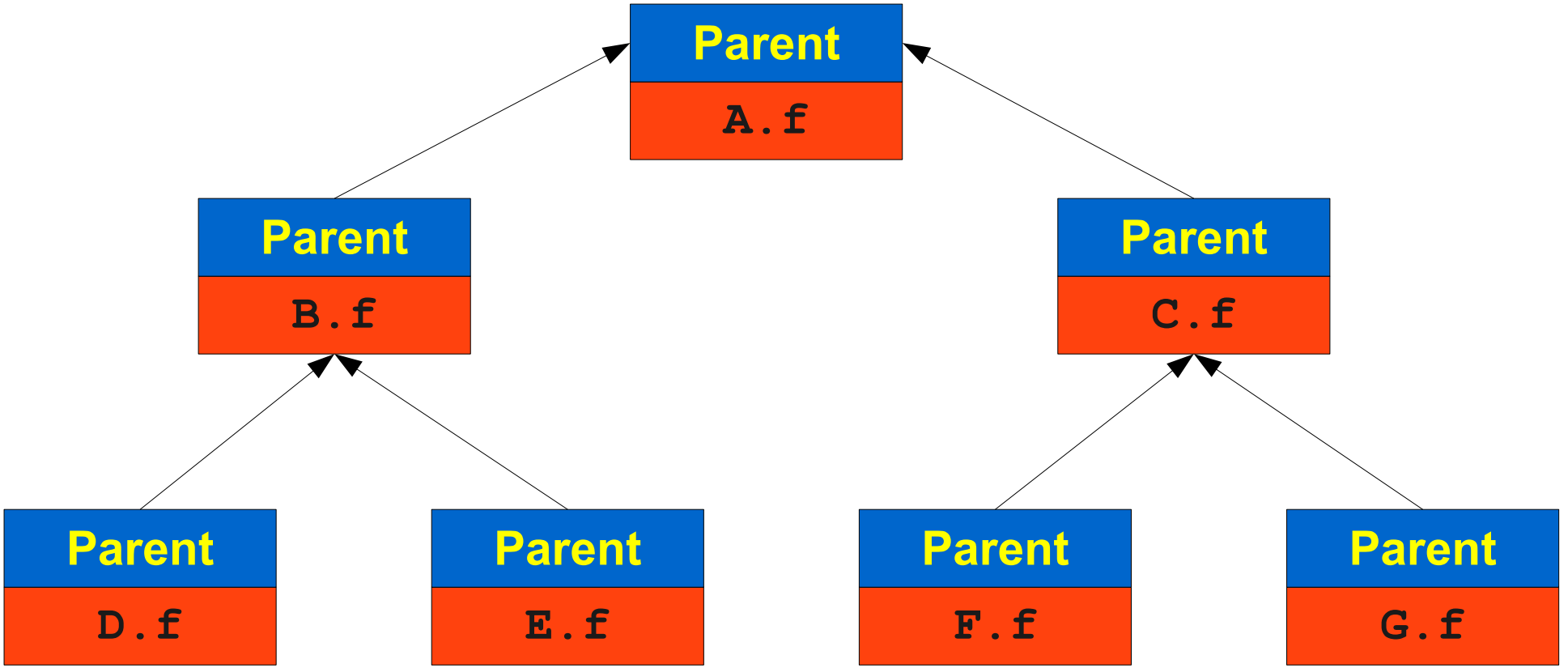
Simple Dynamic Type Checking

- 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.
- Can we make this faster?

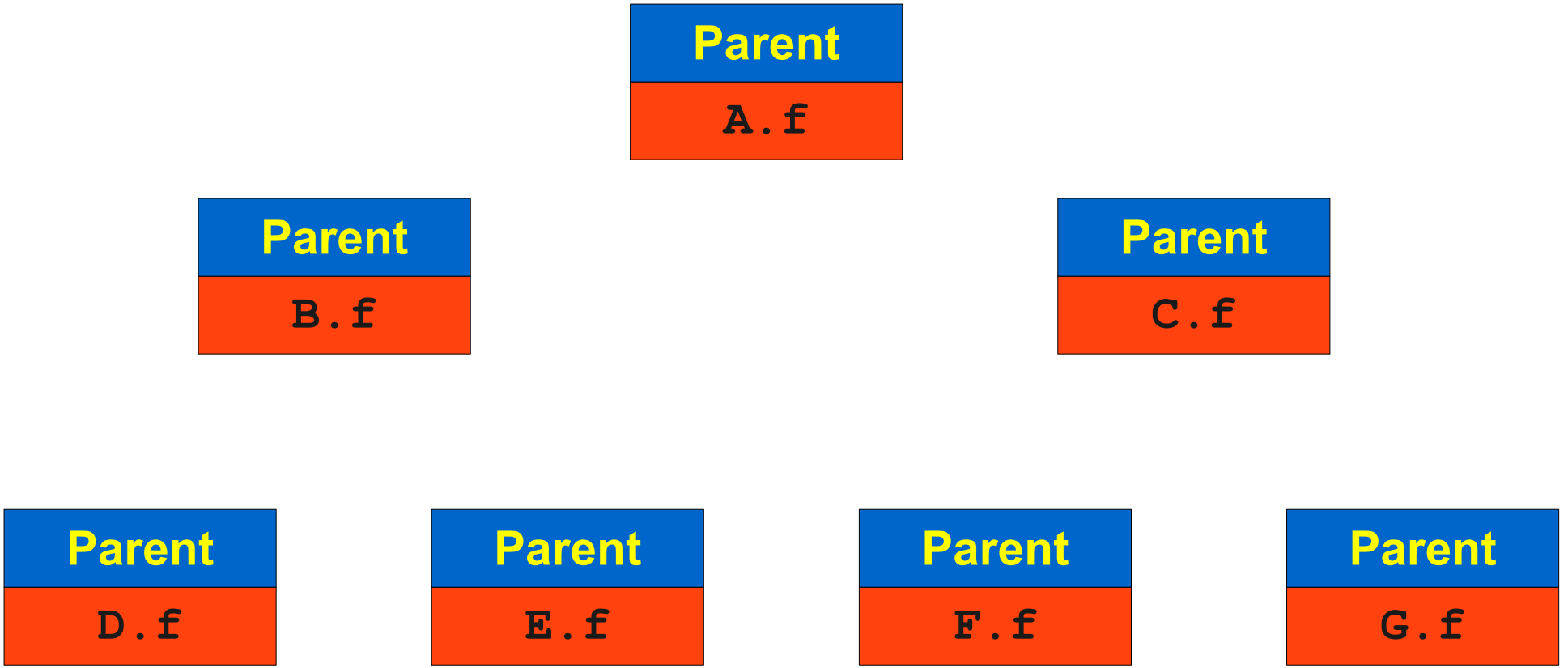
A Marvelous Idea

- There is a fantastically clever way of checking convertibility at runtime in O(1), assuming there are O(1) classes in a hierarchy.
- Assume:
  - There aren't “too many” classes derived from any one class (say, 10).
  - A runtime check of whether an object that is statically of type A is dynamically of type B is only possible if A ≤ B.
  - All types are known statically.

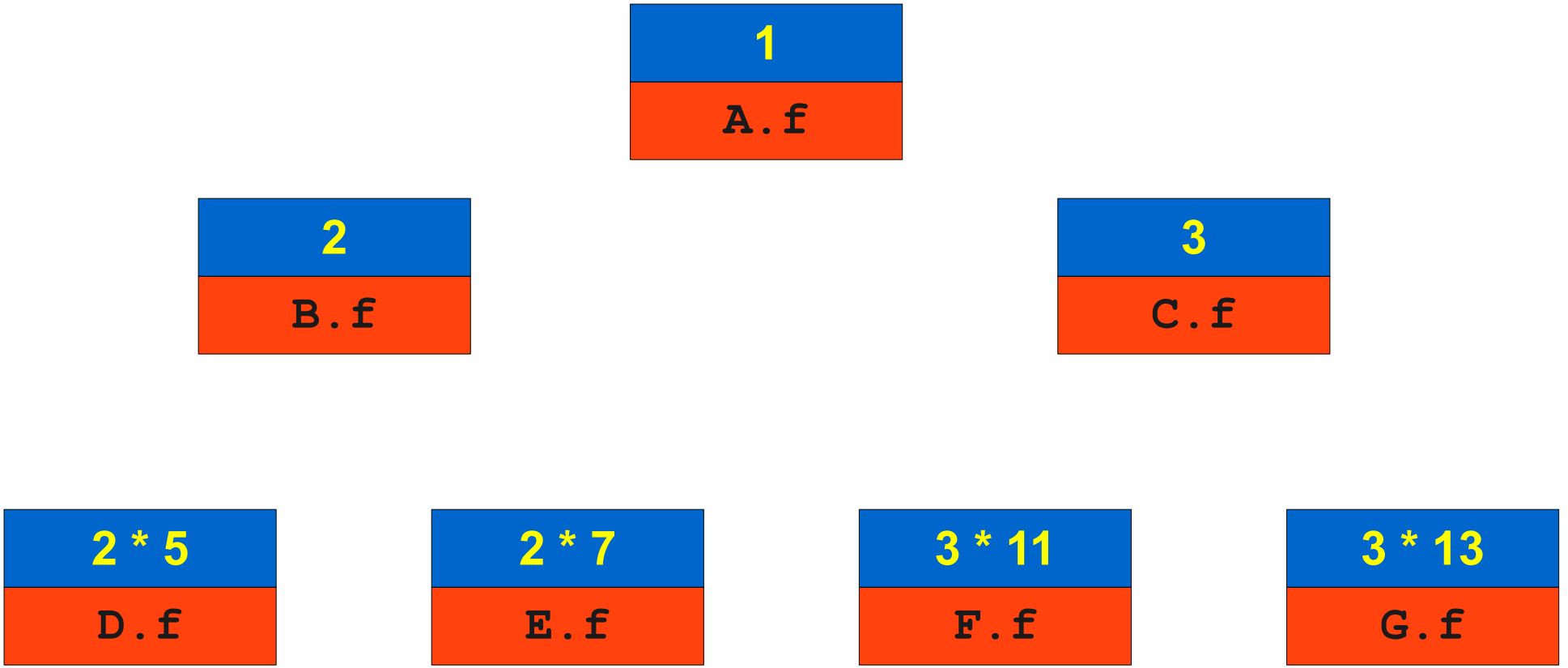
A Marvelous Idea



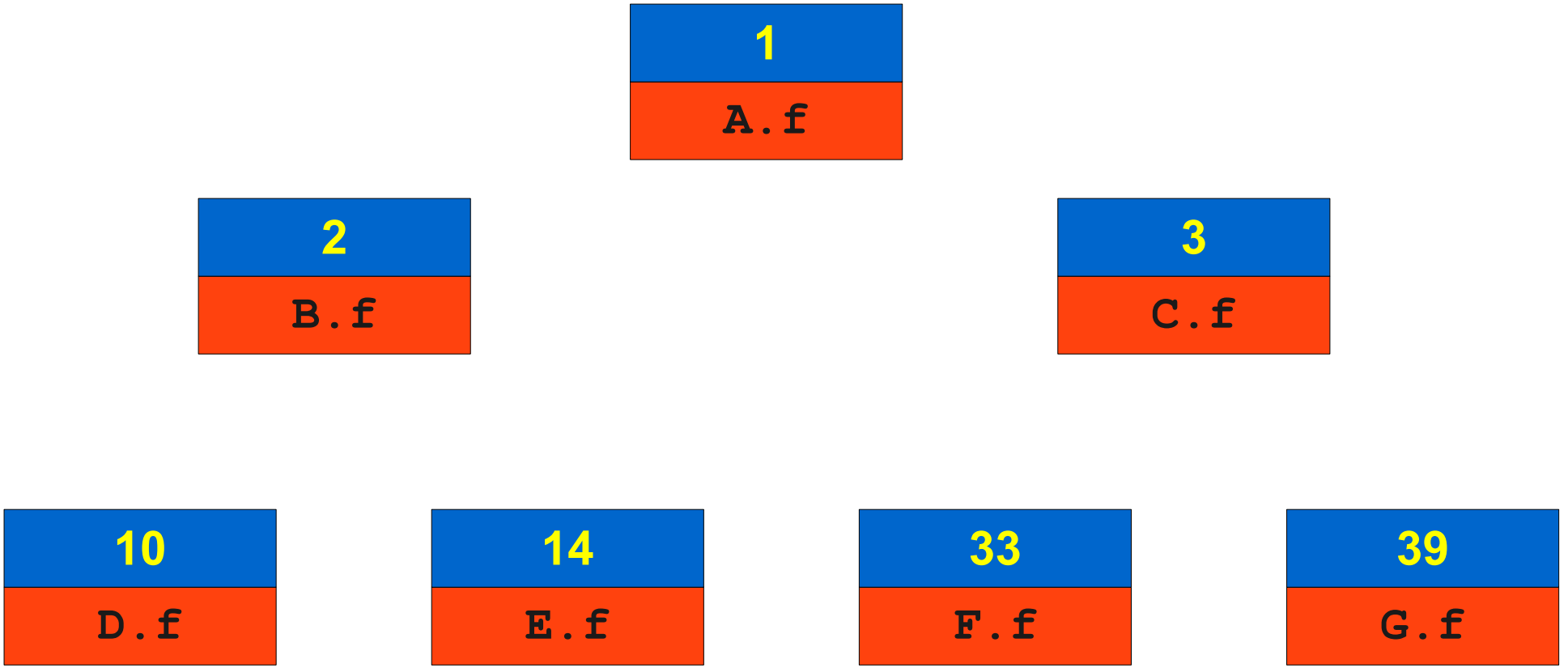
A Marvelous Idea



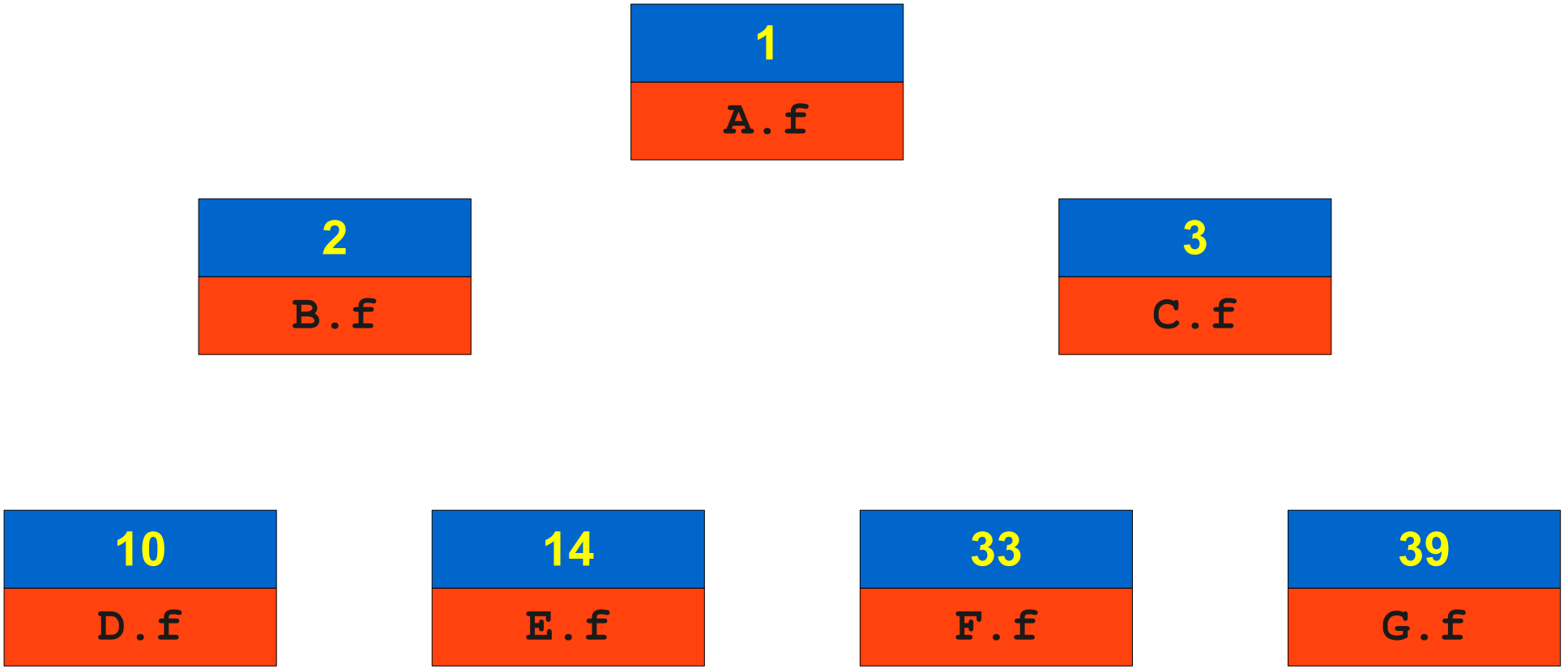
A Marvelous Idea



A Marvelous Idea

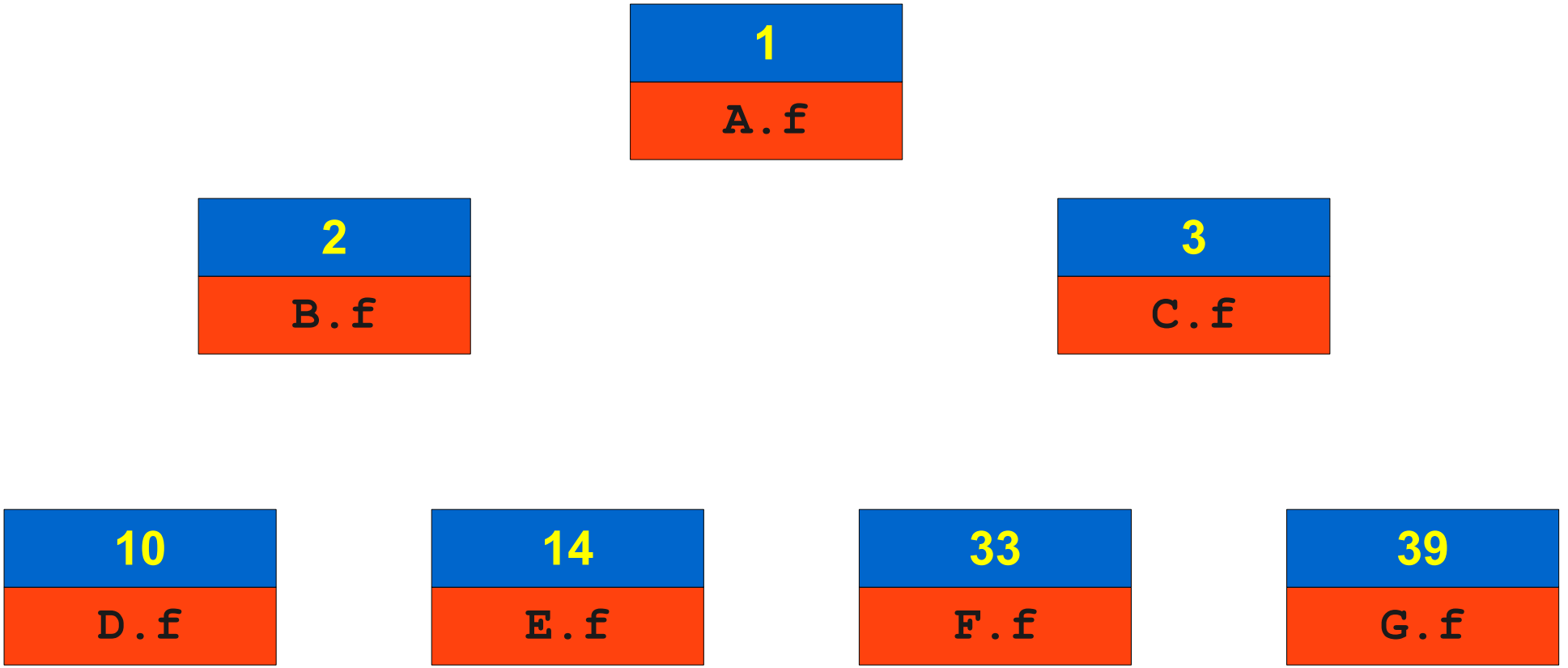


A Marvelous Idea



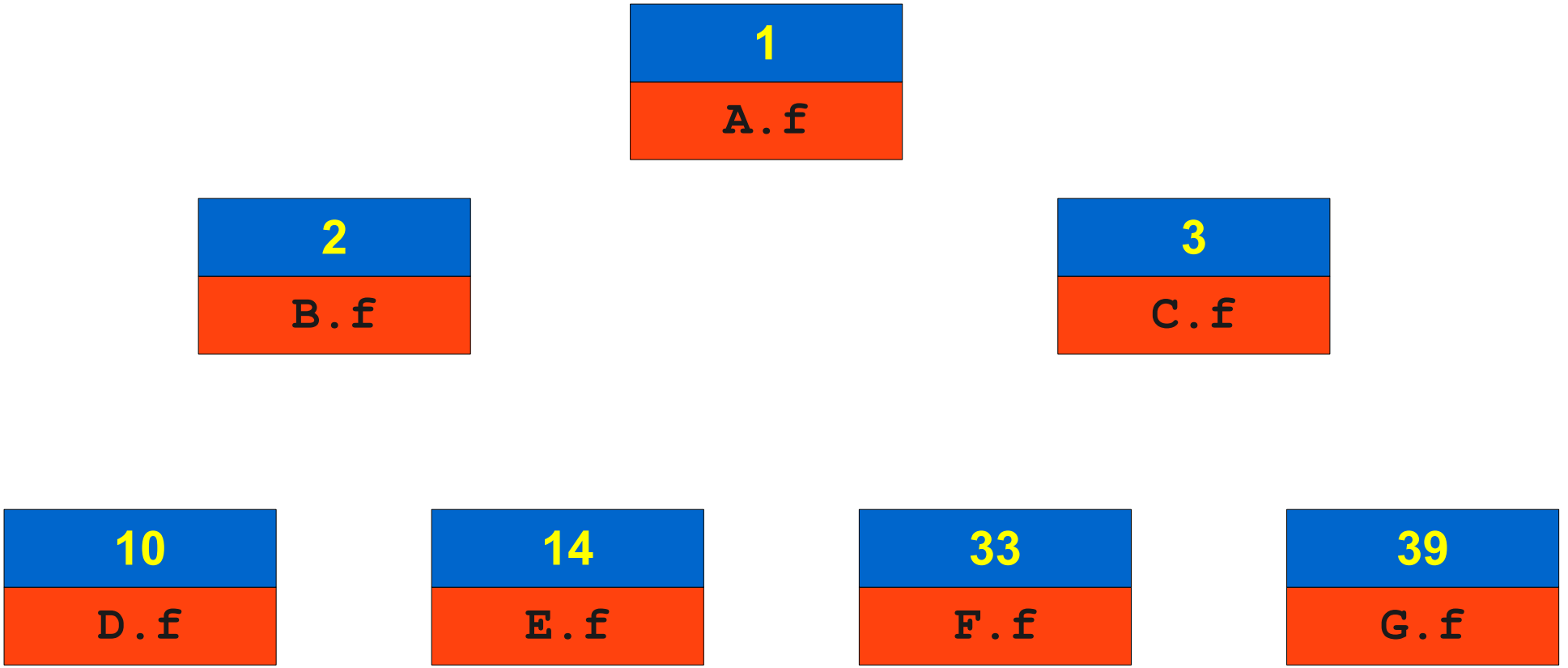

```
A myObject = /* … */
if (myObject instanceof C) {
    /* … */
}
```

A Marvelous Idea




```
A myObject = /* … */
if (myObject instanceof C) {
     /* … */
}
```

A Marvelous Idea




```
A myObject = /* … */
if (myObject->vtable.key % 3 == 0) {
     /* … */
}
```

Dynamic Typing through Primes

- Assign each class a unique prime number.
  - (Can reuse primes across unrelated type hierarchies.)
- Set the **key** of that class to be the product of its prime and all the primes of its superclasses.
- To check at runtime if an object is convertible to type T:
  - Look up the object's key.
  - If T's key divides the object's key, the object is convertible to T.
  - Otherwise, it is not.
- Assuming product of primes fits into an integer, can do this check in **O(1)**.
- Also works with multiple inheritance; prototype C++ implementations using this techinique exist.

Next Time

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