Code Generation III

CONSTRUCTORS AND DATA BOXING

Review: Activation Records

Function Parameters	Return Address	Return Value	Saved Registers	Local Variables	Temporary Values

Activation record or ***stack frame***: memory for each function:

- Machine state.
- Function parameters and return value.
- Local data.

- ***May be empty for small functions.***

Review: Calling Conventions

Responsibilities and ***expectations*** of caller and callee.

- Registers vs. stack.
- Caller or callee saving.
- **Stack alignment**.

- In real life, these are determined by the CPU-OS.
  - Windows conventions are slightly different from Linux. :-(
- Caller and callee ***must agree.***
  - But a single program can use more than one convention.

Review: Object Layout

Superclass fields precede subclass fields.

- Typically ***in definition order***. (Why?)
- Contiguous, with padding for alignment.

Cool fields are ***references***.

Type Tag
Object Size
vtable
Base.field1
Base.field2
Derived.field1
…

Reference Compiler Object Layout

Type Tag: identifies runtime type

- For `case` expressions, object equality.

Object Size:

- Used in `Object.copy()`

vtable: pointer to dispatch table

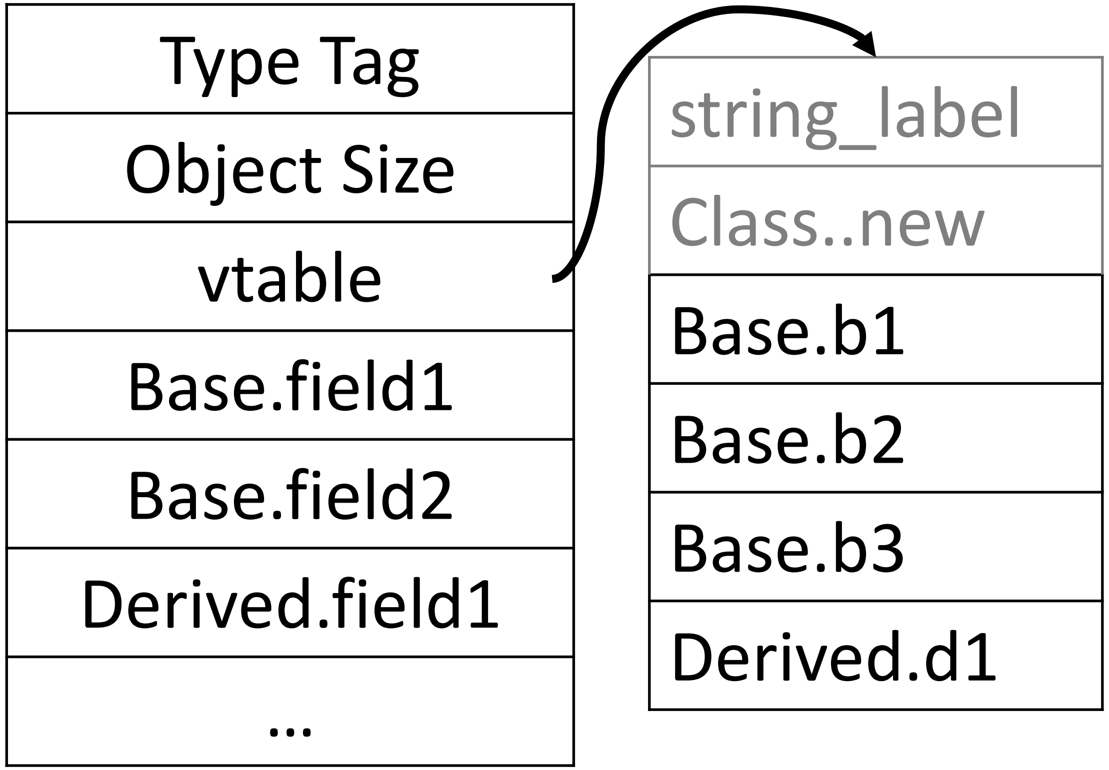
You are free to use a different layout.

Type Tag
Object Size
vtable
Base.field1
Base.field2
Derived.field1
…

Review: Dispatch Tables

Lay out methods like fields

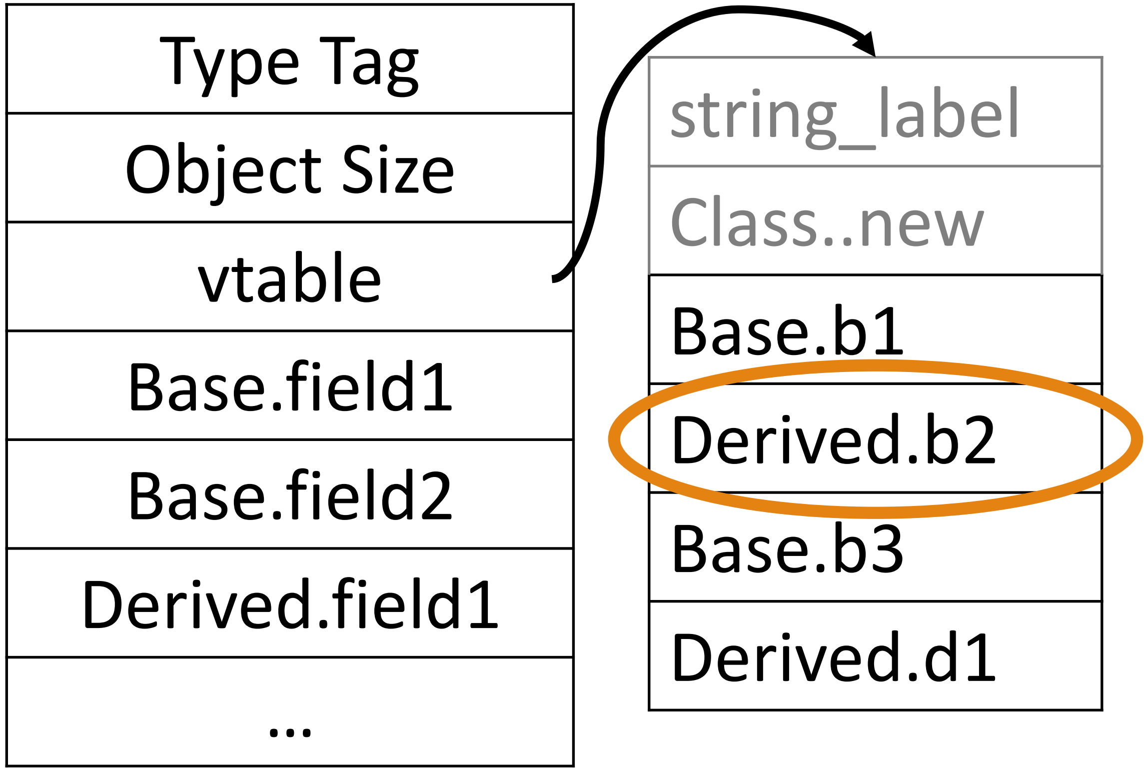
- Base class first.
- Standardized order.



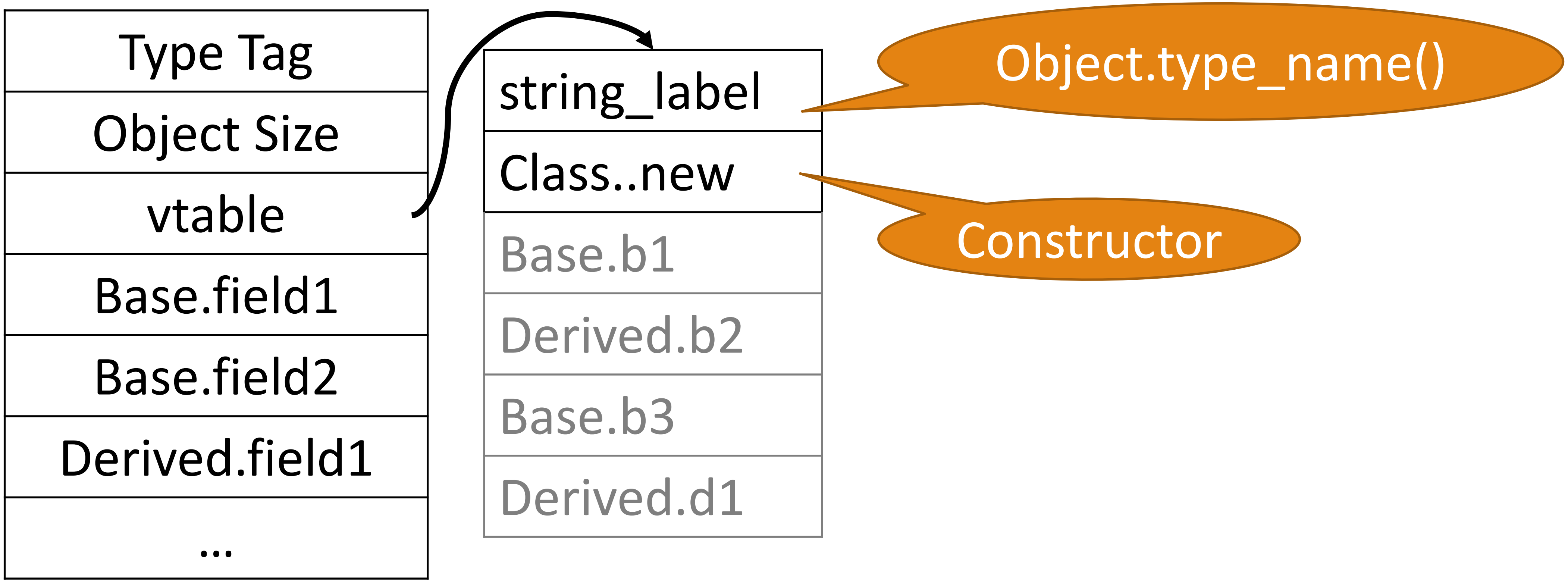
Review: Dispatch Tables

Lay out methods like fields

- Base class first.
- Standardized order.
- Overrides replace base class method.



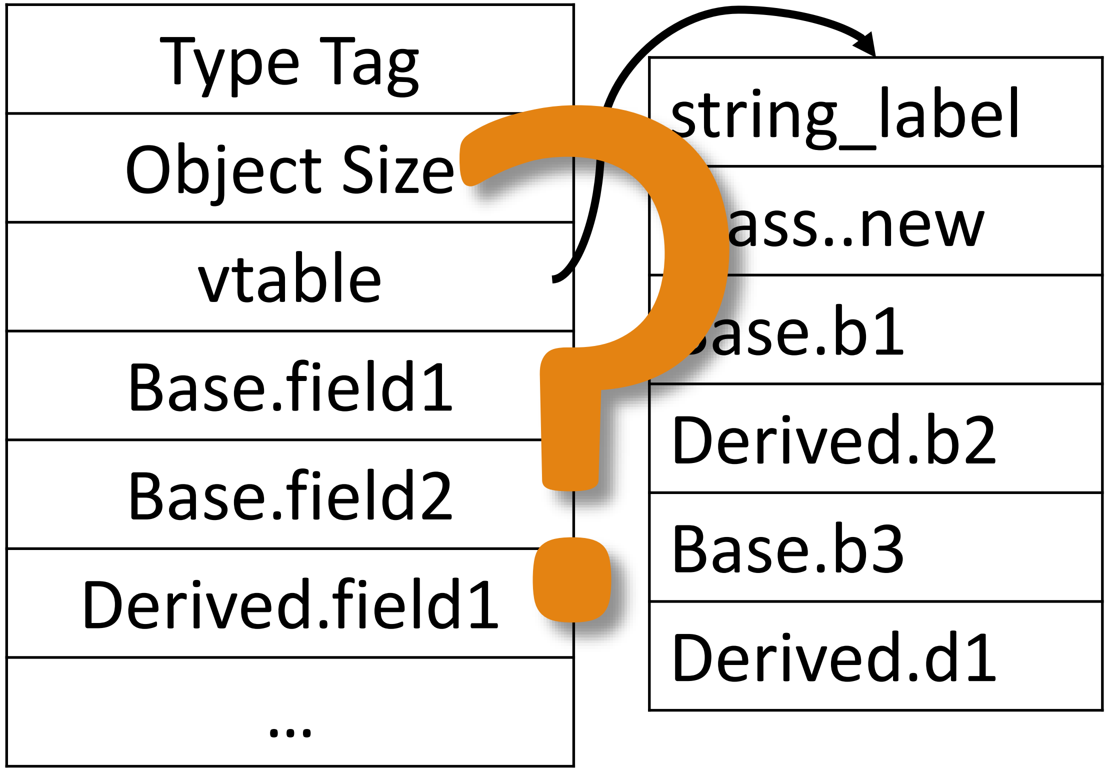
Reference Compiler Dispatch Tables



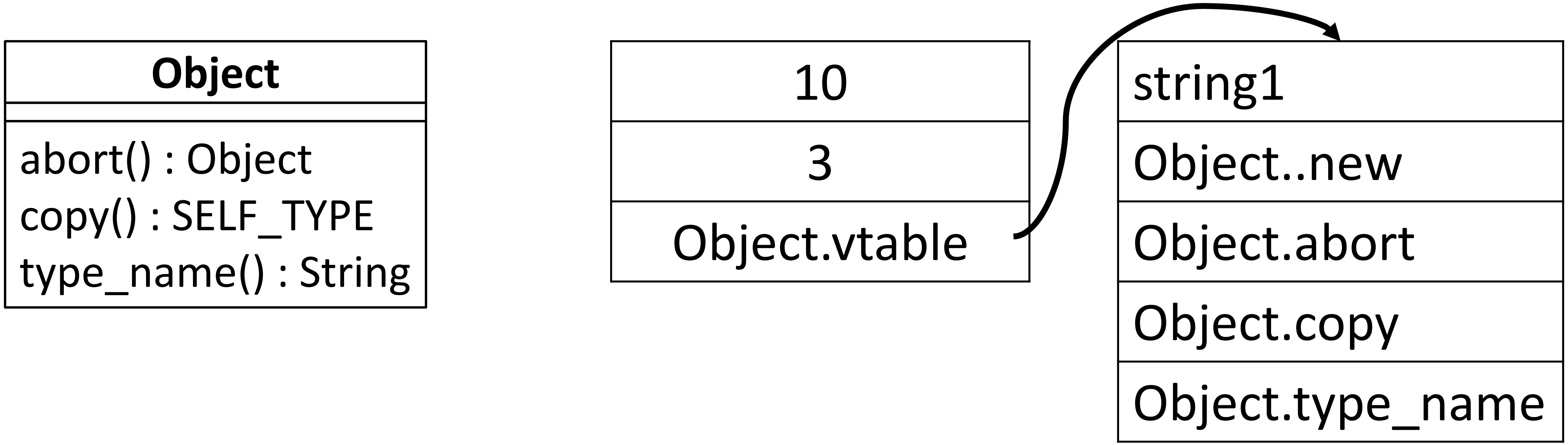
Object Layout Example: Object

Object

abort() : Object
copy() : SELF_TYPE
type_name() : String



Object Layout Example: Object



Constructors

1. Allocate memory.
2. Initialize fields.
   - Compile-time constants.
   - Default initialization.
   - Call user initializers.

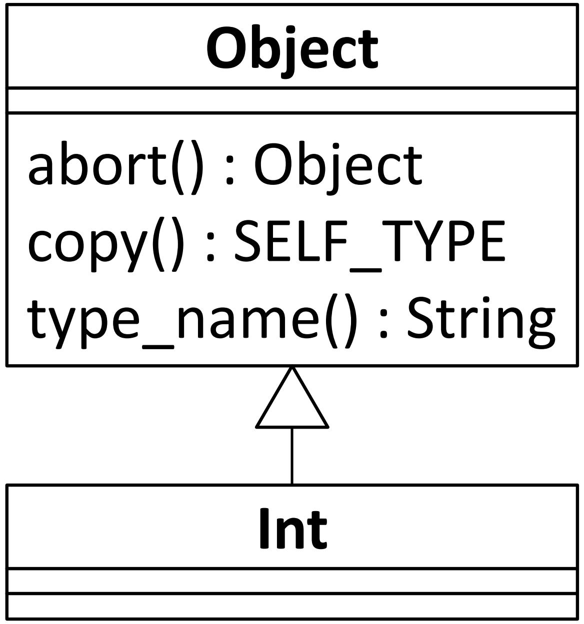
Type Tag
Object Size
vtable
Base.field1
Base.field2
Derived.field1
…

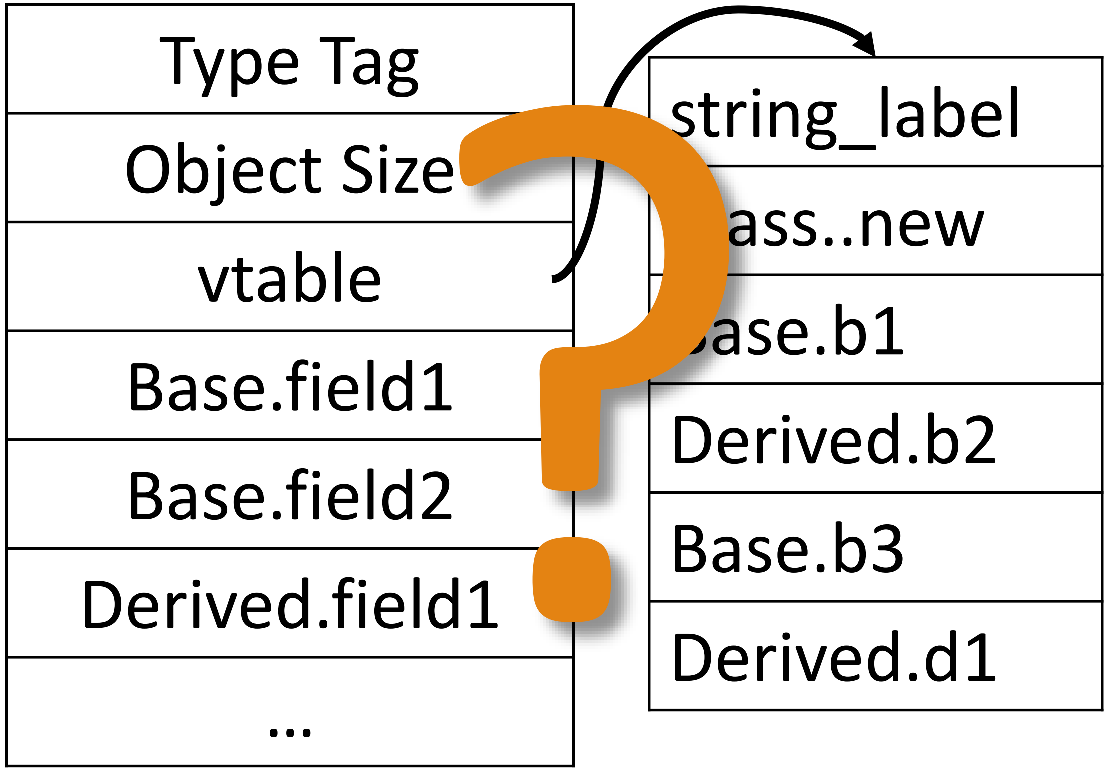
Constructor Example: Object

10
3
Object.vtable

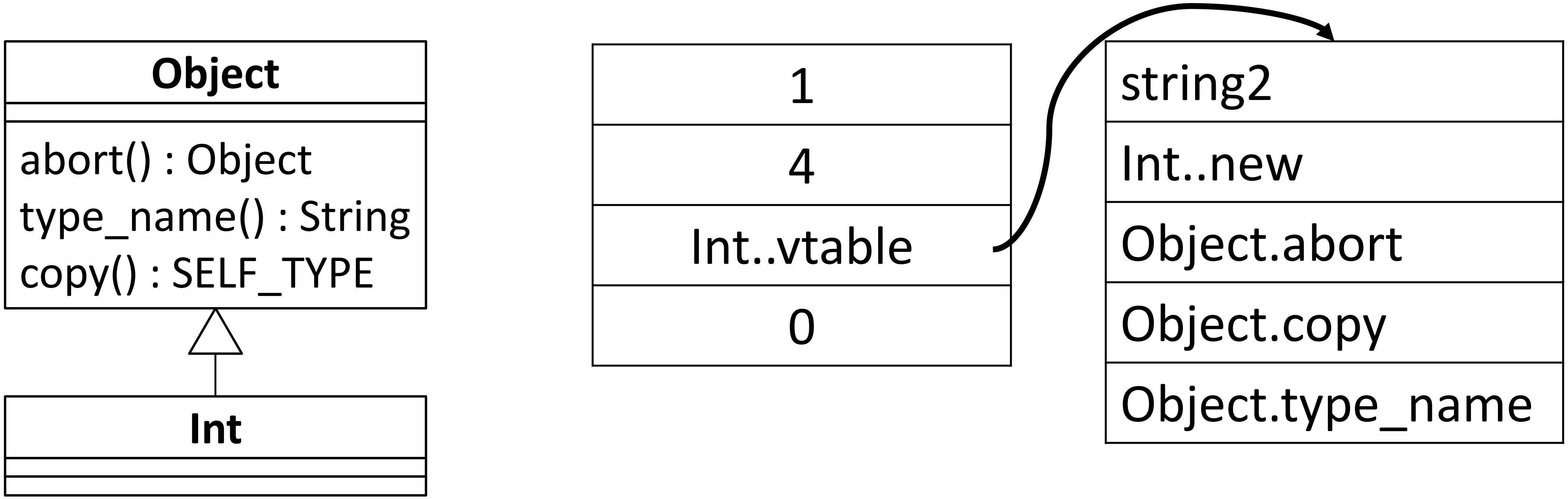
string1
Object..new
Object.abort
Object.copy
Object.type_name

Object Layout Example: Int





Object Layout Example: Int



The New 1 + 2

Ints are now ***boxed***: extra layer of indirection to get at values.

```
;; exp: 1
call Int..new
;; new Int in r1
li r0 <- 1
st r1[3] <- r0
push r1
```

```
;; exp: 2
call Int..new
li r0 <- 2
st r1[3] <- r0
push r1
```

```
;; exp: 1 + 2
pop r1
ld r1 <- r1[3]
pop r0
ld r0 <- r0[3]
add r0 <- r0 r1
call Int..new
st r1[3] <- r0
```

The New 1 + 2

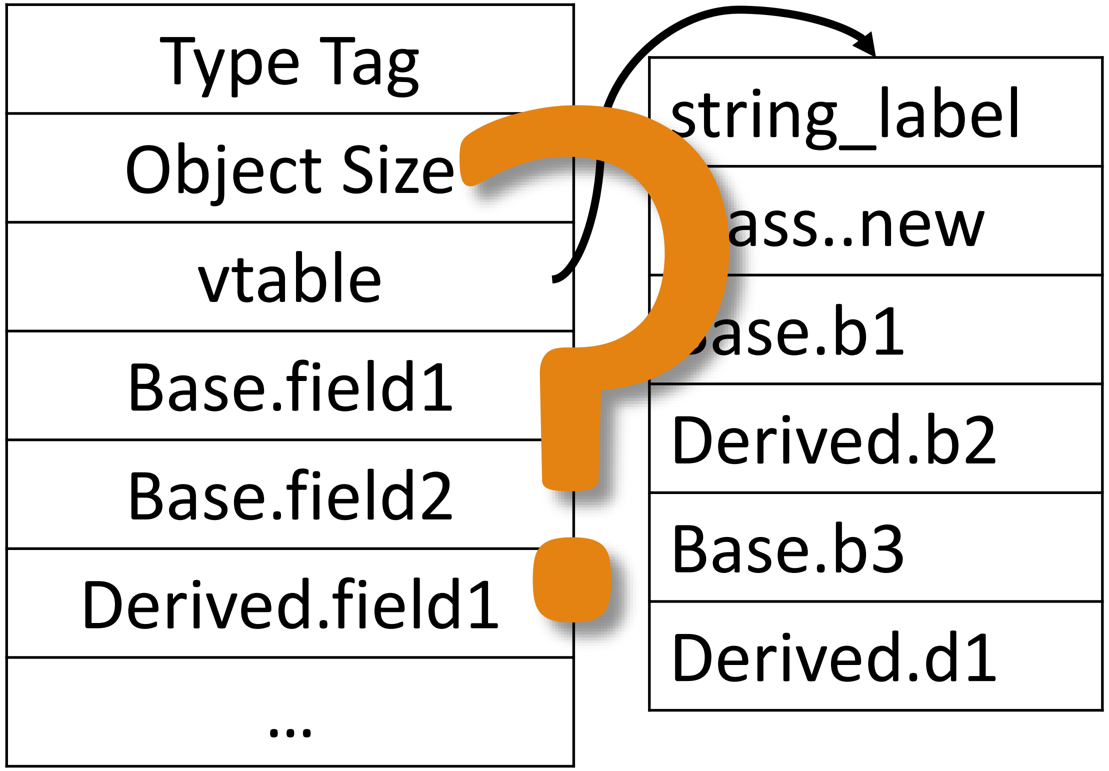
Or we could do it the C++ way: with helper functions.

```
Int.plus:
    push ra
    push r0
    ld r0 <- sp[2]
    ld r0 <- r0[3]
    ld r1 <- sp[3]
    ld r1 <- r1[3]
    add r0 <- r0 r1
```

```
    call Int..new
    ;; new Int in r1
    st r1[3] <- r0
    pop r0
    pop ra
    return
```

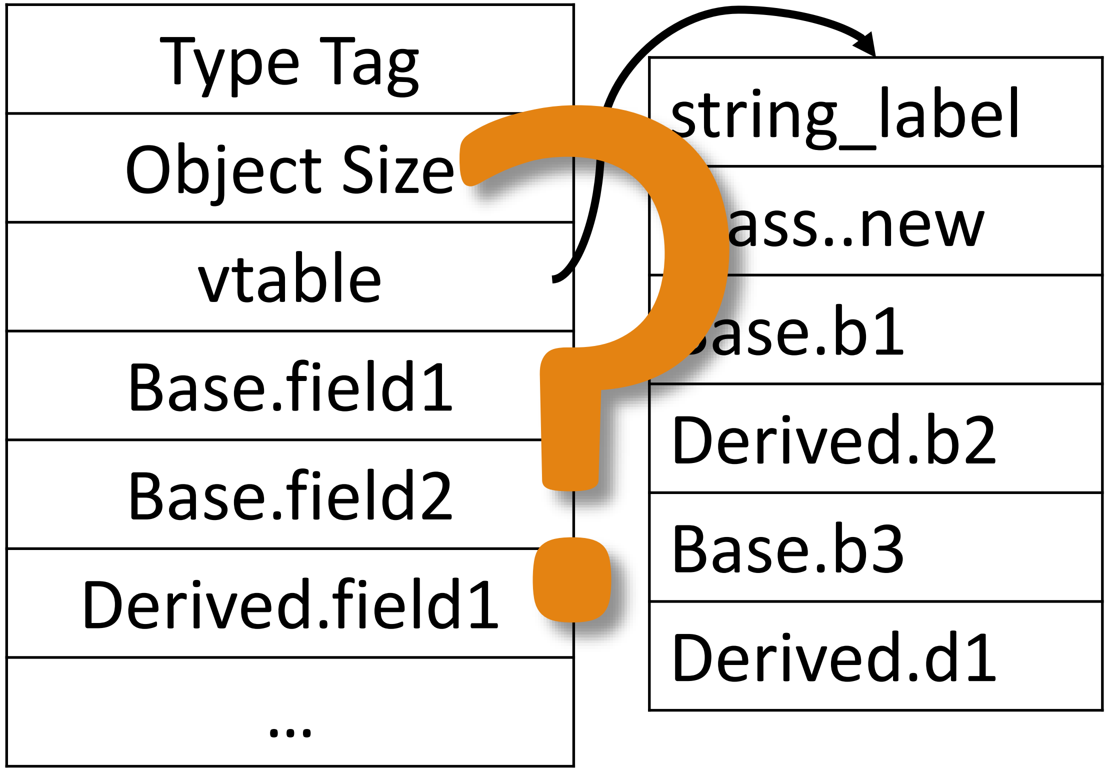
Constructors: Initializers

```
class Foo {
  x : Int <- 1;
  y : Int <- 2;
  dofoo() : Object {…};
};
```



Constructors: Initializers

```
class Bar {
  x : Int <- y + 1;
  y : Int <- x + 1;
  dofoo() : Object {…};
};
```



Object Comparison

Runtime Type	Runtime Type	Comparison
Bool	Bool	False < True
Int	Int	Numeric
String	String	Lexicographic
*	*	Pointer equality

Reference compiler uses type tags:

- Bool = 0
- Int = 1
- String = 3
- Everything else: ?

Why not String = 2?

Case Expressions

Determine closest ancestor of e's ***runtime*** type from {T1, T2, …}.

Error on void.

Error on no ancestor listed.

```
case e of
   x : T1 => e1;
   y : T2 => e2;
   …
esac
```

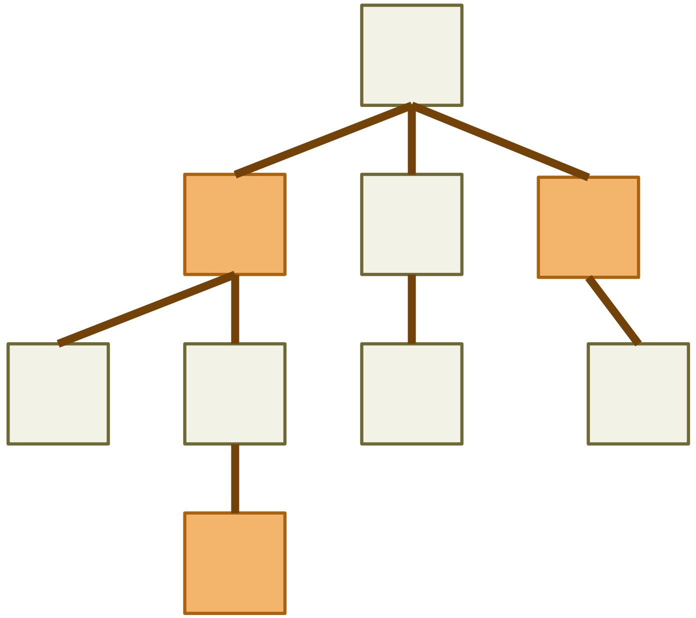
Case Expressions with Type Tags

Initialize target to error label.

Walk type tree depth-first.

For each type:

- If type is in case expression, it becomes new target.
- Emit code to check for type tag and jump to target.



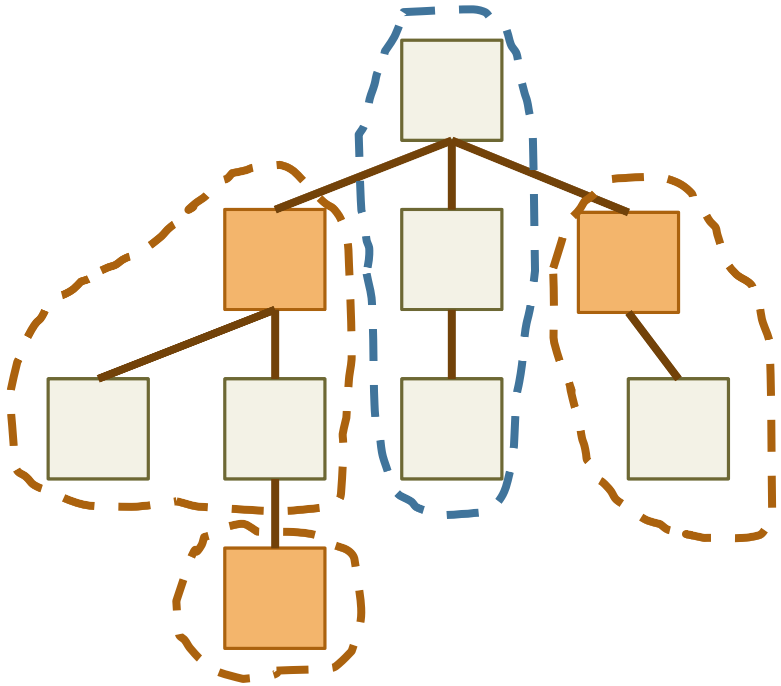
Case Expressions with Type Tags

Initialize target to error label.

Walk type tree depth-first.

For each type:

- If type is in case expression, it becomes new target.
- Emit code to check for type tag and jump to target.



Alternative Implementations

What if we don’t know the complete type tree?

- E.g. classes in separate files.

How else could we implement object comparison?

What about a different object layout?

How do we know which implementation is “best”?

Are We Done?

$$
\begin{array}{rcl}
\mathit{program} & ::= & \llbracket \mathit{class}; \rrbracket^{+} \\
\mathit{class} & ::= & \textbf{class}\ \text{TYPE}\ [\textbf{inherits}\ \text{TYPE}]\ \{\ \llbracket \mathit{feature}; \rrbracket^{*} \} \\
\mathit{feature} & ::= & \text{ID}(\ [\ \mathit{formal}\ \llbracket, \mathit{formal} \rrbracket^{*}\ ]\ ) : \text{TYPE}\ \{\ \mathit{expr}\ \} \\
 & | & \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ] \\
\mathit{formal} & ::= & \text{ID} : \text{TYPE} \\
\mathit{expr} & ::= & \text{ID}\ \texttt{<-}\ \mathit{expr} \\
 & | & \mathit{expr}[@\text{TYPE}].\text{ID}(\ [\ \mathit{expr}\ \llbracket, \mathit{expr} \rrbracket^{*}\ ]\ ) \\
 & | & \text{ID}(\ [\ \mathit{expr}\ \llbracket, \mathit{expr} \rrbracket^{*}\ ]\ ) \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{expr}\ \textbf{else}\ \mathit{expr}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathit{expr}\ \textbf{loop}\ \mathit{expr}\ \textbf{pool} \\
 & | & \{\ \llbracket \mathit{expr}; \rrbracket^{+} \} \\
 & | & \textbf{let}\ \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ]\ \llbracket, \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ] \rrbracket^{*}\ \textbf{in}\ \mathit{expr} \\
 & | & \textbf{case}\ \mathit{expr}\ \textbf{of}\ \llbracket \text{ID} : \text{TYPE} \Rightarrow \mathit{expr}; \rrbracket^{+} \textbf{esac} \\
 & | & \textbf{new}\ \text{TYPE} \\
 & | & \textbf{isvoid}\ \mathit{expr} \\
 & | & \mathit{expr} + \mathit{expr} \\
 & | & \mathit{expr} - \mathit{expr} \\
 & | & \mathit{expr} * \mathit{expr} \\
 & | & \mathit{expr} / \mathit{expr} \\
 & | & \text{\textasciitilde}\,\mathit{expr} \\
 & | & \mathit{expr} < \mathit{expr} \\
 & | & \mathit{expr} <= \mathit{expr} \\
 & | & \mathit{expr} = \mathit{expr} \\
 & | & \textbf{not}\ \mathit{expr} \\
 & | & (\mathit{expr}) \\
 & | & \text{ID} \\
 & | & \text{integer} \\
 & | & \text{string} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

Final Thoughts

Check out `cool --asm` and `cool --x86` output.

- Built-in methods, object comparison, etc. are all defined.

Remember: you ***do not*** have to make the same choices for calling conventions, object layout (but you can if you want).