Object Orientation

PLC

J. Heliotis

1. Issues and Concepts

PLC

J. Heliotis

New Issues in OO Languages

- Types and classes
- Creation/destruction
- Self
- Scope
- Feature access
- Inheritance
  - Polymorphism

Is a class a type?

- 'Type' can be
  - a set of values (denotational)
  - a set of legal operations (abstraction)
  - a representation (constructive)
- 'Class' adds
  - a set of operation implementations

Prototype-based Languages

- Prototype-based
  - Self
  - [JavaScript]
- Objects contain everything they need,
  - including a pointer to another object to which all operations not implemented locally are *delegated*.

Other Interpreted Languages

- Classes ⊆ Objects
  - Smalltalk
  - Python
  - [Java]
- Objects point to their classes.
  - Classes may have *metaclasses*.
- Classes also point to their superclasses.

Fully Compiled Languages

- Classes don't exist at run time
  - C++
    - (Well, there is RTTI.)
- Objects are `structs`, with
  - Associated functions that contain an extra parameter with a struct pointer
    - Possibly hidden in source code
  - Indirect function address lookup tables for polymorphism

Types and Classes

- Object descriptions are classes.
- Classes are an extension of the *record* concept...

Pre-OO and OO
What does this tell us?


```
// ADT Manager
struct Foo {
  T1 a;
  T2 b;
};
void init(
  struct Foo *f ) {
  // …
}
void m(
  struct Foo *f ) {
  // …
}
```

```
// True class
class Foo {
  T1 a;
  T2 b;
  Foo() {
    // …
  }
  void m() {
    // …
  }
};
```

Module: Precursor to Class


```
module Foo {
  struct Data {
    T1 a;
    T2 b;
  };
  Data *create() {
    // …
  }
  void m(Data *d) {
    // …
  }
}
```

```
import Foo;

Foo.Data *x = Foo.create();

Foo.m(x);

print( x.a, x.b );
```

close to Python, except...?

Modula, Mesa, Ada, CLU (see later slide)

First: Midterm 3 Summary

Classes As Extensions of ADT Managers


```
# File tree.py
class BSTNode(object):
    __slots__ = ("value","left","right")

def makeNode(v):
    n = BSTNode()
    n.value = v
    n.left = None
    n.right = None
    return n
```

continued...

Classes As Extensions of ADT Managers


```
def add( n, newValue ):
    if newValue < n.value:
        if n.left is not None:
            add( n.left, newValue )
        else:
            n.left = makeNode( newValue )
    else:
        if n.right is not None:
            add( n.right, newValue )
        else:
            n.right = makeNode( newValue )
```

CLU [MIT, Barbara Liskov et al]


```
complex_number = cluster is
   add, subtract, multiply, ...
   rep = record [
      real_part: real,
      imag_part: real ]
   add = proc ... end add;
   subtract = proc ... end subtract;
   multiply = proc ... end multiply;
   ...
end complex_number;
```

Creation/Destruction

- Old

```
obj = malloc( sizeof( struct Foo ) );
Foo_init( obj );
```

- OO

What's happening here?

```
obj = new Foo();
```

Allocates memory

Initializes data

Self: A Call's Target Object

- An extra argument is added to each instance subroutine's parameter list.
- It is known as
  - **`self`** (Smalltalk, Python*, Ruby, Ada-95)
  - **`this`** (Simula, C++, Java, C#)
  - **`Current`** (Eiffel)

Scope

- Recall: What is the scope of **`x`** inside **`m`**?
  - global?
  - file?
  - subroutine local?
- Technically, **`x`** is not a variable.
- It's a field inside the record named by **`this`**.

```
class C {
    int x;
    void m() {
      x = 5;
    }
}
```

```
class C {
    int x;
    void m(C this) {
      this.x = 5;
    }
}
```

Again: invoking an operation

- C obj;
  obj.m(x, y);
  ⇓
- struct C obj;
  m(obj, x, y);

Does an OO language do anything more than adjust syntax?

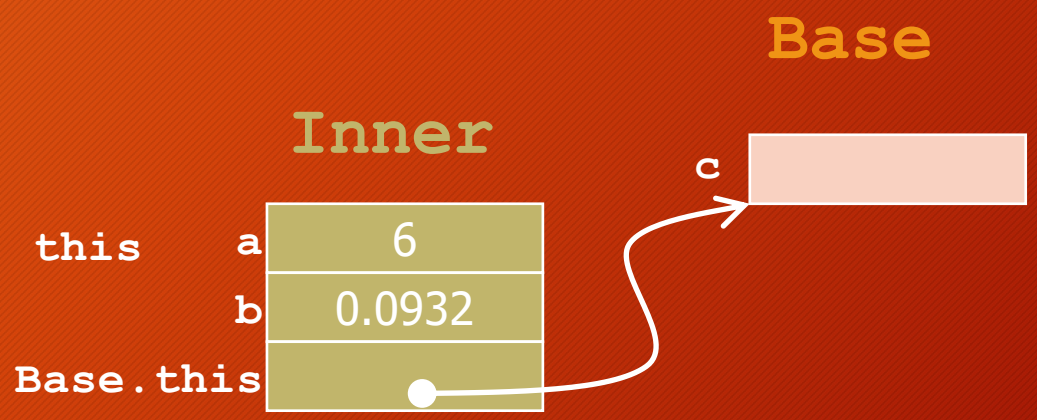
Feature Access

- 3- or 4-level (C++, Java, C#)
  - **`public`**
  - **`protected`** (loophole!)
  - **`private`**
  - **`internal`**
- by class (Eiffel)
  - **`feature`** – public
  - **`feature { SOME_CLASS }`**
    – restricted to SOME_CLASS & descendants
  - **`feature { NONE }`**
    – accessible only to this class's routines (not via **`Current`**.)

- **`friend`**

Side Trip
Nested Classes


```
class Base {
  boolean c;
  class Inner {
    int a;
    float b;
  }
}
```

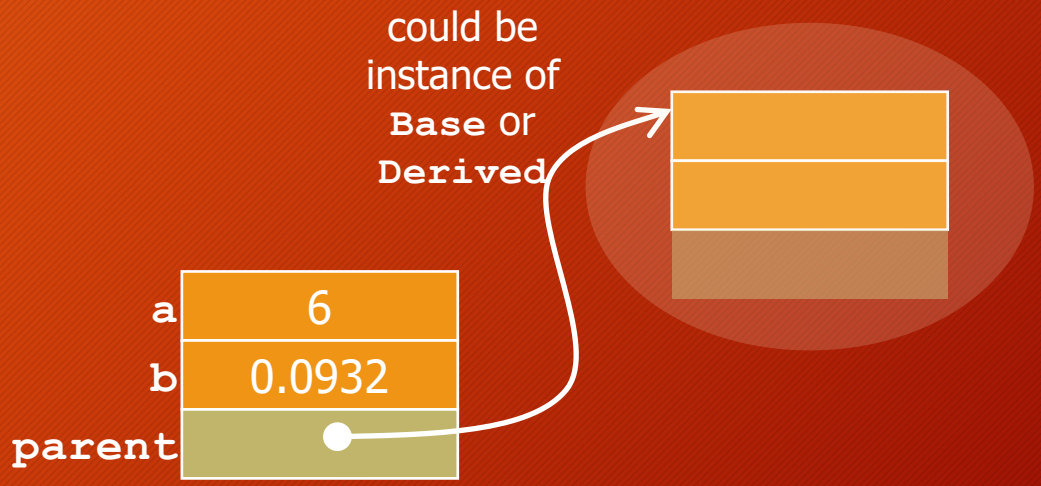


- Instance of nested class requires instance of outer class.

Inheritance

- Structure of subclass is appended to structure of superclass.
- Offsets to fields in new structure are easy to compute:

```
class Base {
  int a;
  float b;
}
class Derived: Base {
  Base *parent;
}
```



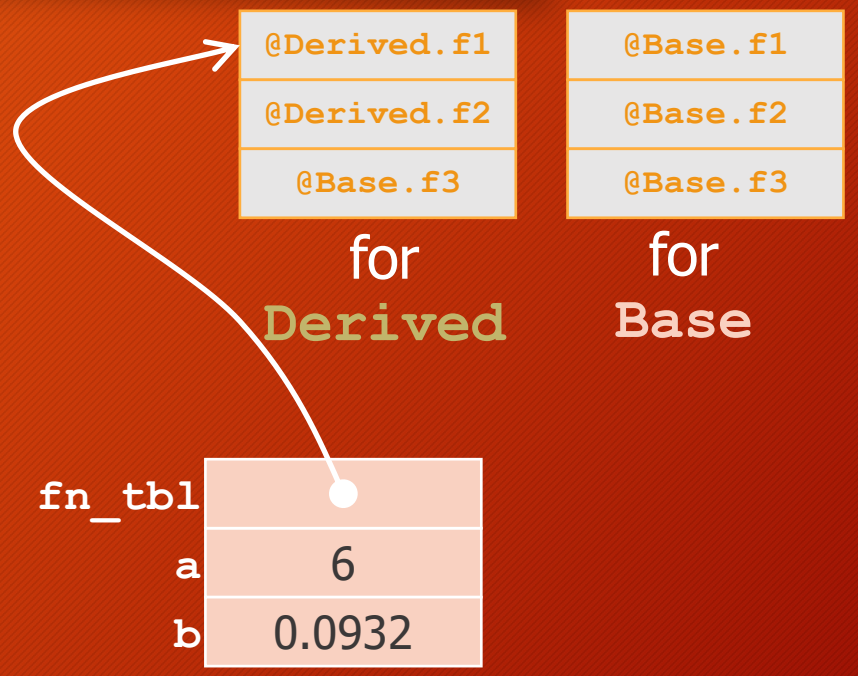
a: @objectPointer
b: @objectPointer+4
parent: @objectPointer+8

Polymorphism

- Two approaches:
  - Virtual function table
  - Class objects

Function Table

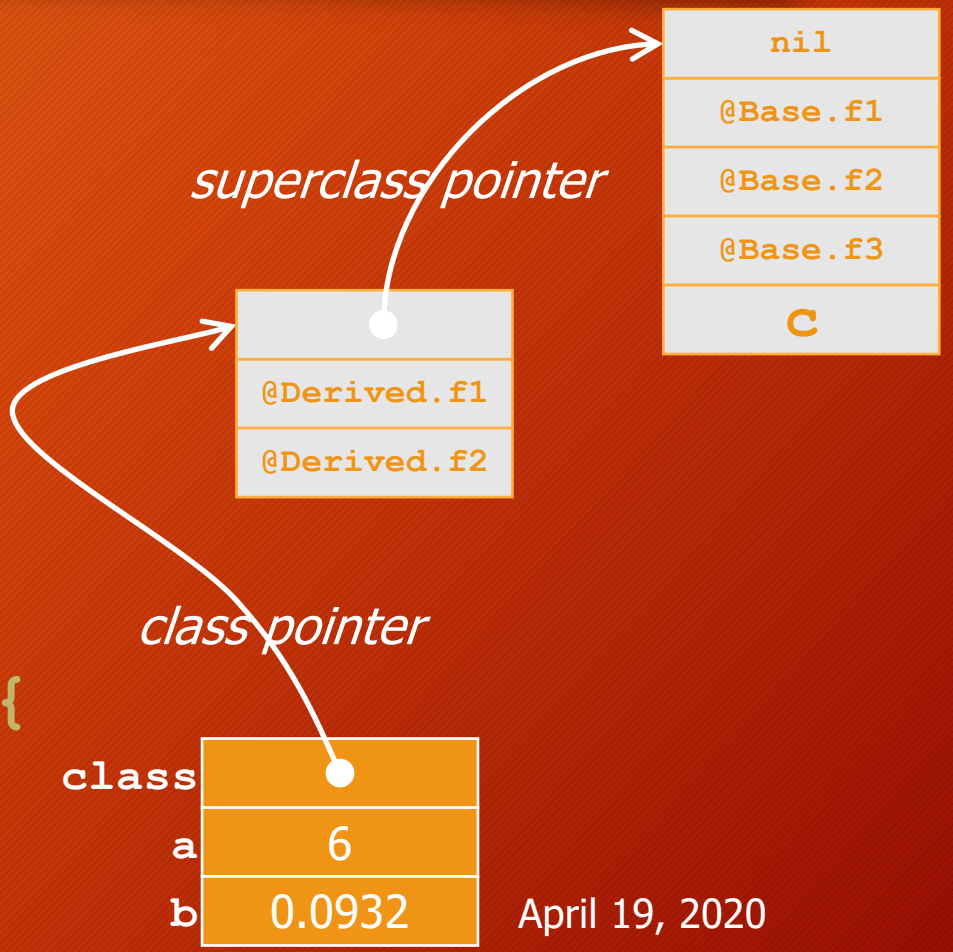

```
class Base {
   int a;
   float b;
   void f1(){…}
   void f2(){…}
   void f3(){…}
}
class Derived: Base {
   void f1(){…}
   void f2(){…}
}
```



C++ compilers typically use this for virtual functions.

Class Object
(closer to plcc language)


```
class Base {
   int a;
   float b;
   static int c;
   void f1(){…}
   void f2(){…}
   void f3(){…}
}
class Derived: Base {
   void f1(){…}
   void f2(){…}
}
```



Multiple Inheritance in Python

- There is a tree representing the ancestors of a class.
- A linearization of the classes in that tree is constructed.
- When given a method name, each class in that linear list is checked for the presence of the method.
- (attributes?)

Multiple Inheritance in Eiffel

- If a new class inherits another class more than once, only one instance of the other class's attributes appears in an instance of the new class.
- Identically named attributes (fields) are automatically merged together.
- Identically named routines must be dealt with.
- Actions specifiable in inheriting a class
  - Rename a feature
  - Select a feature as the polymorphic version of something inherited from multiple ancestors
  - Undefine a version of a feature from an ancestor class (effectively selects the one left)

What Goes Where?

A. static variables
B. static methods
C. instance variables
D. instance methods
E. inheritance relationships

Classes should contain:

Objects should contain:

*class-based=="`static`" in Java/C++

2. OBJ

The Object-Oriented Language using PLCC

But First: Other Stuff in OBJ


```
--> {display 19; display 84; newline}
1984
nil
--> {display# 19; display 84; newline}
19 84
nil
--> display# [1,2,3,[4,5]]
[1,2,3,[4,5]] [1,2,3,[4,5]]
--> display# "abc"
[97,98,99] [97,98,99]
```

The Basics

- We build objects using environments.
- A class's environment includes (as we learned)
  - static variables
  - methods
  - superclass relationship
- An object's environment includes (as we learned)
  - instance variables
  - other built-in stuff to navigate the hierarchies

Fossum's First Example


```
define c1 =
    class % extends an unnamed class
        field x
        field y end
define c2 =
    class extends c1
        field z
        field x end
define obj1 = new c1
define obj2 = new c2
<obj1>set x = 3
<obj2>set x = 5
<obj1>x % evaluates to 3
```

Things to Note


```
define c1 =
    class
        field x
        field y end
define c2 =
    class extends c1
        field z
        field x end
define obj1 = new c1
define obj2 = new c2
<obj1>set x = 3
<obj2>set x = 5
<obj1>x % evaluates to 3
```

- Classes are special environments.
- Objects are special environments.
- But classes are not objects.
- Up until now we could not treat an environment as a first class entity.
  - We can assign an environment to a variable.
  - We can write an expression that evaluates to an environment.

<env>exp

"Evaluate the expression *exp* in the context of the environment *env*."

Environment Qualification Examples


```
--> define x = 1
x
--> define outer = @
outer
--> define inner = let x = 2 in @
inner
--> <outer>x
1
--> <inner>x
2
```

Fossum's Example Again
Note use of objects as environments


```
define c1 =
    class % extends an unnamed class
        field x
        field y end
define c2 =
    class extends c1
        field z
        field x end
define obj1 = new c1
define obj2 = new c2
<obj1>set x = 3
<obj2>set x = 5
<obj1>x % evaluates to 3
```

Class expressions can go in the angle brackets, too.

Using environments in methods


```
--> define Pair
        class
            field x
            field y
            method sum =
                proc() +(x,y)
        end

Pair
--> define p = new Pair
p
```

```
--> set <p>x = 2
2
--> <p>x
2
--> <p>set y = 13
13
--> <p>.sum()
15
--> .<p>sum()
15
```

ValRefs!

so does it matter which one we do?

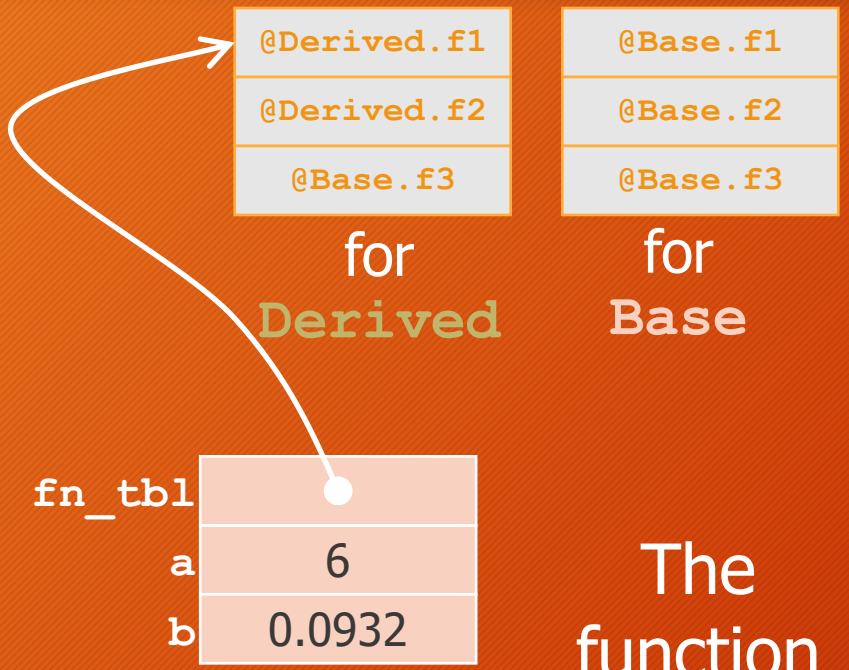
oh yeah

and you thought it was hard to remember the dot…

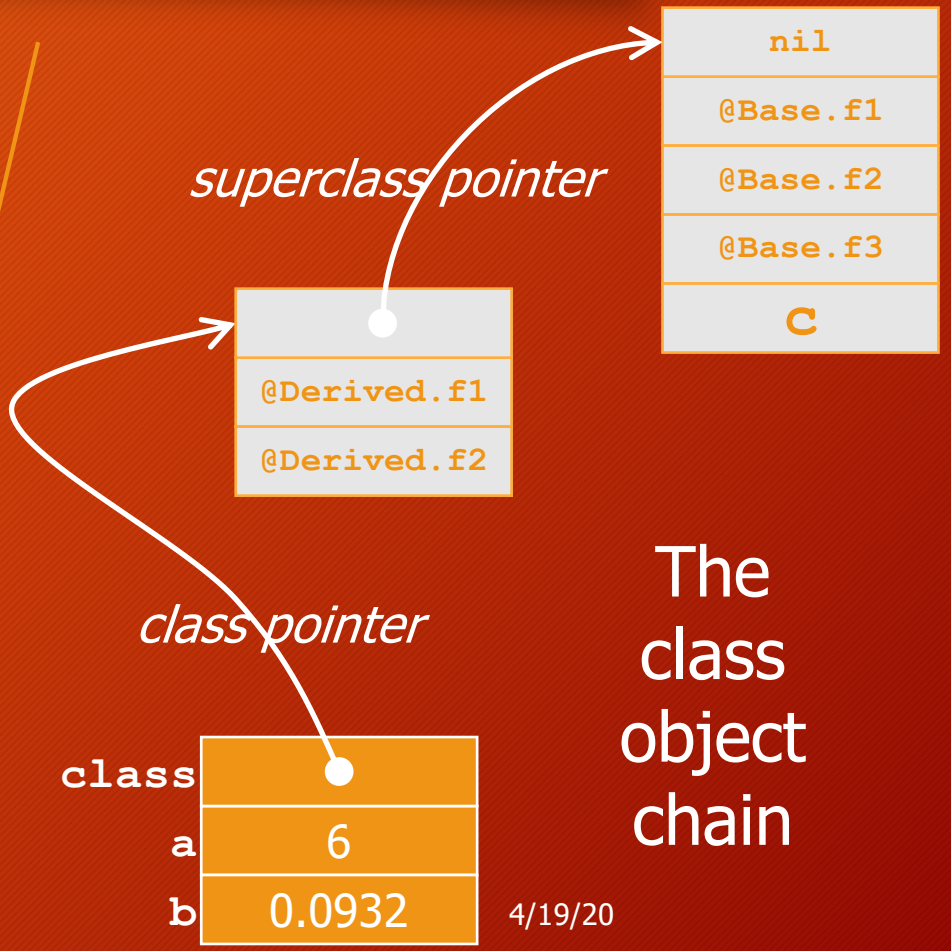

```
--> define Num = class field x method plus = proc( a ) +( x, a ) end
--> define x = 10
x
--> define n = new Num
n
--> <n>set x = 2
2
--> <n>.plus( x )
4
--> .<n>plus( x )
12
```

- Which one should we normally use?

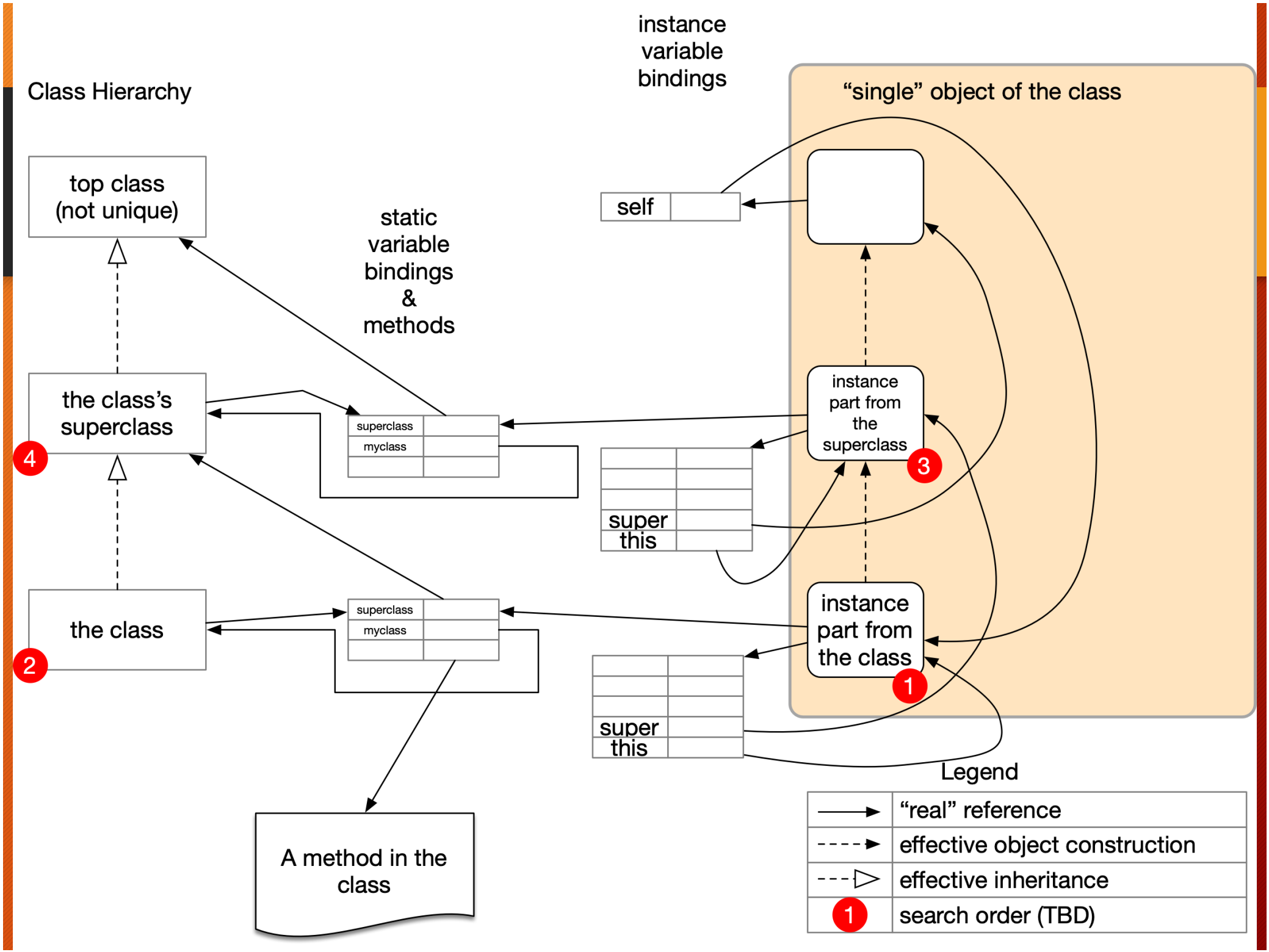
Reminder: OOP Design Approaches



The function table



The class object chain


Class Hierarchy
top class
(not unique)
the class's
superclass
the class
A method in the
class
static
variable
bindings
&
methods
superclass
myclass
superclass
myclass
instance
variable
bindings
self
super
this
super
this
"single" object of the class
instance
part from
the
superclass
instance
part from
the class
1
2
3
4
Legend
"real" reference
effective object construction
effective inheritance
search order (TBD)

Class/Superclass Example


```
define Super = class
                 static x = 10
                 static y = 11
                 field a
                 field b
                 method init = proc( m, n ) {
                     set a = m;
                     set b = n;
                     self
                 }
             end
```

Super
$x $y a b
init(m, n)

Class/Superclass Example


```
define Sub = class extends Super
                 static x = 20
                 static z = 22
                 field a
                 field c
                 method init = proc( p, q, r ) {
                     .<super>init(*(p,2),q);
                     set a = p;
                     set c = r;
                     self
                 }
             end
```

Super
$x $y a b
init(m, n)

Sub
$x $z a c
init(p,q,r)

Class/Superclass Example

Super
$x $y a b
init(m, n)

```
--> define obj = .<new Sub>init( 100, 130, 160 )
obj
--> <obj>a
100
--> <obj>b
130
--> <obj>c
160
--> <obj>super
object
--> <<obj>super>a
200
--> <<obj>super>b
130
--> <<obj>super>c
no binding for c
```

Sub
$x $z a c
init(p,q,r)

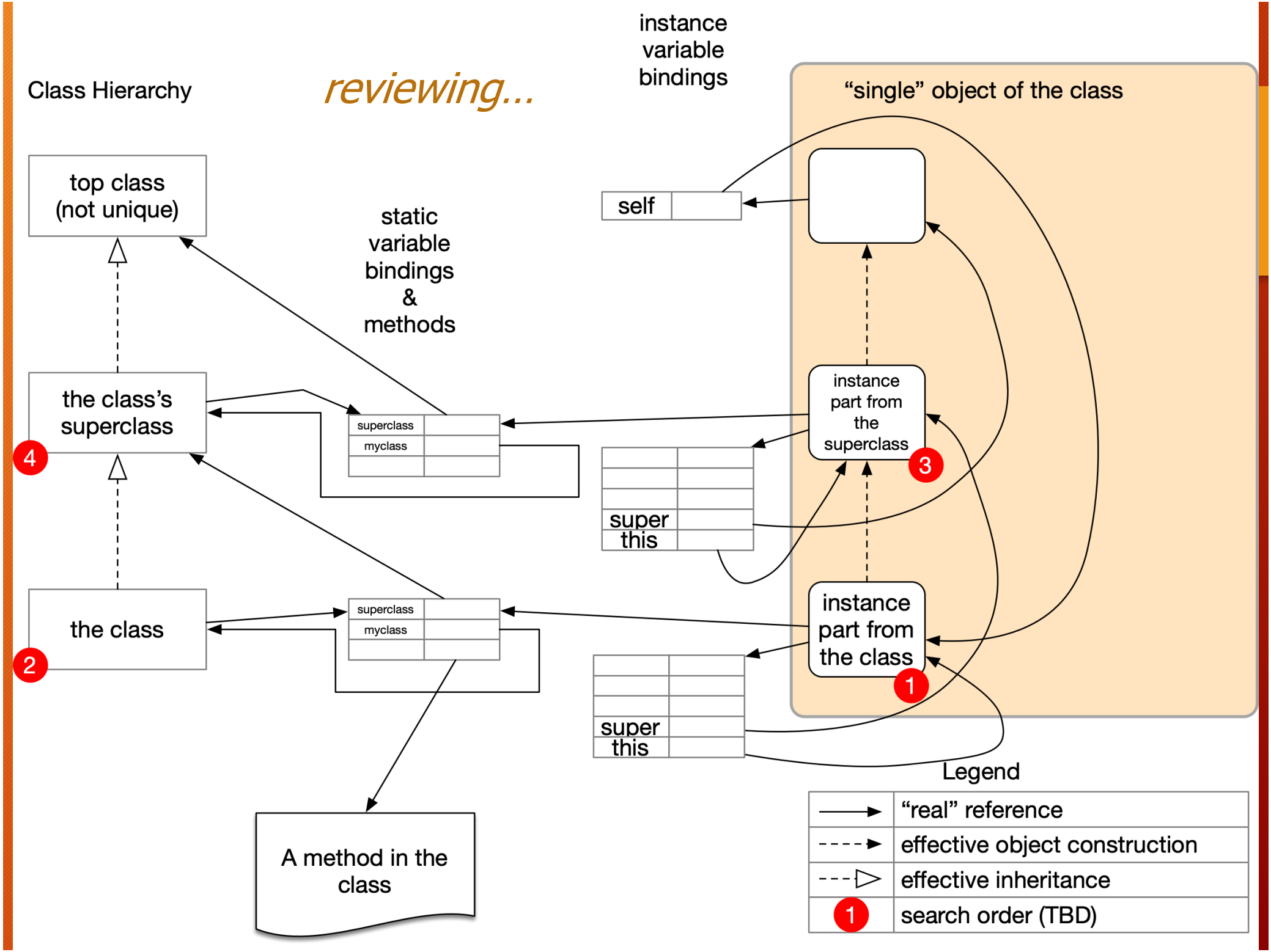
Simple Examples

- accumulator
- circ-5ths
  - based on music idea of circle of fifths
  - The next "fifth up" is actually 7 notes up from the current one.

D-------A-------E--------B-------F#----

When an Object is Created

- Object is allocated.
- Object gets an environment whose bindings refer to the fields.
- Object also gets connected to the static environment of its class.
- A separate object *part* is created for every class in the ancestor chain!
  - Those object parts are also chained together.
- Search order: wheres-waldo.obj


Class Hierarchy
reviewing...
instance variable bindings
"single" object of the class
top class (not unique)
static variable bindings & methods
self
the class's superclass
4
superclass
myclass
instance part from the superclass
3
super
this
the class
2
superclass
myclass
instance part from the class
1
super
this
A method in the class
Legend
"real" reference
effective object construction
effective inheritance
1
search order (TBD)

When an Object is Created

- Class variables
  - `myclass`: refers back to class
  - `superclass`
- Instance variables
  - `this`: the object environment in which the current method is running (may not be bottom of object chain)
  - `super`: the environment of the part of the object chain above this
  - `self`: the actual object, i.e. the bottom of the object chain (defined at the top of the chain)

Summary

- What are the requirements for a language to be object-oriented?
- What are some OO language designs?
- (How does OBJ do object orientation?)
  - Important: how environment concept is applied
- What are the syntax and semantics of OBJ?
- For homework, investigate other new stuff.
  - Lists
  - Characters, strings, output