Classes and Objects

Compared to most object-oriented programming languages such as Java and C++, our classes and objects are *first class* – that is, they are values that can be created at any time and in any environment, they can be bound to variables, they can be passed as arguments to procedures, and they can be returned as values of procedure applications.

A *class* is an expressed value that captures a collection of variables (called *fields*) and procedures (called *methods*) and that serves as a template to create expressed values called *instances* of the class. The instances are called *objects*, to distinguish them from classes. You can think of a class as a factory that identifies how to churn out an arbitrary number of instances (objects) of the class.

Language OBJ

All classes of an `OBJ` program belong to a *class hierarchy*, which is tree structure with an unnamed class at the root of the tree and with program-created classes at the other nodes of the tree.

In the class hierarchy, a class `X` that occurs as a child node of class `Y` in the class hierarchy is called a *subclass* of `Y`, and `Y` is called a *superclass* of `X`. In this case, we also say that `X` *extends* `Y`.

When an object of class `X` is instantiated, instances of each of the classes that lie on the path from `X` to the root of the tree are created, and the combination of all those instances is considered as the resulting object.

If `Y` is the superclass of `X`, then an object `x` created from class `X` "contains" an object `y` created from `Y`. The object `y` is called the *parent* of `x`, and likewise `x` is called the *child* of `y`.

A class may also have `static` variables whose values are shared among all instances of the class.

If `x` is an object and `f` is a field, the expression `<x>f` evaluates to the value bound to `f` in object `x`. In languages like C++ and Java, this would be written instead as `x.f`.

Language OBJ (continued)

Consider the following example.

```
define c1 =
  class % extends the unnamed top-level class
    field x
    field y
  end

define c2 =
  class extends c1
    field z
  end

define o2 = new c2

<<o2>super>set x = 3
<o2>x % evaluates to 3
```

In this example, c1 is a class and c2 is a subclass of c1. Object o2 is an instance of c2 and <o2>super is an instance of c1.

Language OBJ (continued)

We define the concrete and abstract syntax of classes and objects. We also add the additional reserved word `nil` to our language, which represents a separate expressed value not shared by any other data type. When used in conditional expressions, `nil` is considered false. (The only other such value is the `IntVal` of zero.)

```
<exp>:NilExp     ::= NIL
                 NilExp()
<exp>:ClassExp   ::= CLASS <ext> <statics> <fields> <methods> END
                 ClassExp(Ext ext, Statics statics, Fields fields, Methods methods)
<ext>:Ext0       ::=
                 Ext0()
<ext>:Ext1       ::= EXTENDS <exp>
                 Ext1(Exp exp)
<statics>        **= STATIC <VAR> EQUALS <exp>
                 Statics(List<Token> varList, List<Exp> expList)
<fields>         **= FIELD <VAR>
                 Fields(List<Token> varList)
<methods>        **= METHOD <VAR> EQUALS <proc>
                 Methods(List<Token> varList, List<Proc> procList)
<exp>:NewExp     ::= NEW <exp>
                 NewExp(Exp exp)
<exp>:EnvExp     ::= LANGLE <exp>vExp RANGLE <exp>eExp
                 EnvExp(Exp vExp, Exp eExp)
```

Language OBJ (continued)

Every `Exp` expression evaluates to a `Val` object, and a `ClassExp` is no exception, so evaluating such a `ClassExp` (*i.e.*, calling its `eval` method) returns a `Val` object. Looking only at the syntax, it seems reasonable that such an object has a superclass (the `ext` [for `EXTENDS`] part), a set of static identifiers and asociated values, a set of field names, and a set of method names and associated procedures.

We define a `StdClass` object in the file `class`, which extends the `Val` class. (Actually, it extends `ClassVal` class which in turn extends the `Val` class.) Evaluating a `ClassExp` expression returns an instance of `StdClass`.

[**Warning**: In this chapter we implement classes and objects in our source language (`OBJ`), using classes and objects in our implementation language (Java). This can be confusing. For example, a `ClassExp` evaluates to an object in Java that represents a class in our source language. Be sure that you have a clear understanding of the context in which the terms “class” and “object” are being used in the following discussion.]

Language OBJ (continued)

A `StdClass` object (we're in the implementation language, Java, now) has the following instance variables:

```
public ClassVal superClass;
public Fields fields;
public Methods methods;
public Bindings staticBindings;
public Env staticEnv;
```

The `superClass` variable is a reference to a `ClassVal` object that is the superclass of this class. The `staticBindings` variable is a reference to the the bindings of the static variable names to their RHS values. The RHS values are evaluated in the current static environment. (More about this later.)

The `fields` and `methods` variables are references to the `Fields` and `Methods` objects that capture the class field names and method names and procedures – these are not evaluated yet.

The `staticEnv` variable is a reference to the static environment of this class: it starts out extending the static environment of the superclass with an empty `Bindings` object named `staticBindings`. New bindings are added to the `staticBindings` list using the variable definitions in the `statics` parameter of the `StdClass` constructor. Each static LHS identifier is bound to the value of its RHS, where the RHS expression is evaluated in the current static environment. These bindings are created in order (first to last) as in top-level `define`s.

Language OBJ (continued)

Two predefined identifiers are inserted initially into the list of static bindings: one binds the identifier `myclass` to (a reference to) this class itself, and another binds the identifier `superclass` to (a reference to) the superclass of this class.

If the class expression does not have an `extends` part, its superclass defaults to the unnamed “parentless” class (a static `EnvClass` Java object) whose static environment is the top-level program environment – so top-level variables that have been `define`d are visible. In this way, all of the RHS expressions in the `static` definitions of such a class have access to the other static bindings in the class itself as well as to the top-level bindings.

A class expresion that specifies an explicit superclass (using `extends`) has an environment that extends the static environment of that superclass, and – going up the superclass chain – has access to the bindings in all of the superclass static environments as well as to the top-level bindings.

If a class expression has a static definition for a variable that also appears as a static variable in a superclass, that definition shadows the superclass variable. We disallow duplicate LHS variable names in a given class expression’s static definitions. We also disallow static redefinitions for `myclass` or `superclass`.

Language OBJ (continued)

Although counter-intuitive, objects are actually simpler than classes, because an object is essentially a wrapper for an instance of `Env`!

An `ObjectVal` is a Java class that extends the `Val` class. It has a single instance variable:

```
public Env objectEnv;
```

The `new` operator in our source language takes a class expression and returns a Java `ObjectVal` instance that essentially couples the `static` bindings of the class with bindings for the class fields and methods.

Since our language does not define an explicit constructor in class expressions, we initially bind object fields to (a reference to) `nil`.

The method variable names in the class definition are bound to procedure closures that capture the environment that includes bindings for the class `static` variables (from the `staticBindings` field), along with bindings for the fields as described above. The method closures are created as in `letrec`, so they can refer to themselves recursively. As with static and field definitions, we disallow duplicate method variable names.

Language OBJ (continued)

Before we build an object from a base class, we first build an object that is an instance of the superclass of the base class. This superclass object has its own environment, namely `objectEnv`. We then extend this superclass object environment by adding bindings to the statics, fields, and methods of the class, and use this extended environment to create the instance of the base class.

Since creating the superclass object may itself involve creating an instance of *its* superclass, object creation continues up the class hierarchy until the top-level `EnvClass` class is found, at which point there is no further superclass object to create.

At the top of the chain of superclass objects, we add the identifier `self` to the environment of this top-level superclass object, binding it to a reference to the (original) base class object being created. Since all of the objects created by going up the superclass chain have environments that ultimatelly extend the top-level object, all of these objects can refer to the base class object being created using the `self` field identifier. (In Java, we do the same thing using `this` instead of `self`.) Methods declared in superclasses that refer to `self` will "see" the base class object, allowing for dynamic dispatch of method calls, an important feature of object-oriented languages. We call this binding of `self` a *deep* binding.

As each object is created up the superclass chain, we insert three predefined fields into its list of field bindings: `this`, `self`, and `super`. We bind the field identifier `self` to the base object being created, which is the same as described above. We bind the field identifier `this` to the object being created at the particular point in the superclass chain (we call this a *shallow* binding). And we bind the field identifier `super` to the superclass object. The code for creating ghese bindings is shown on Slide 5.16.

We disallow duplicate field names in class definitions. We also disallow field names that duplicate the predefined identifiers `this`, `self`, and `super`.

Notice that an object can see all of the `static` bindings up the superclass chain, but that if a `static` variable is bound to a procedure (or some other value that captures an environment), the procedure captures only the static environment of the class and cannot "see" any of the fields or methods – including the `self` and `this` identifiers – in its environment.

Language OBJ (continued)

A an instance of the Java `EnvClass` class has one instance variable:

```
public Env staticEnv;
```

Here is the code for the `EnvClass` constructor:

```
public EnvClass(Env env) {
    // the static environment of this class
    staticEnv = env;
}
```

The `EnvClass` defines a static `envClass` Java object that is a “singleton” instance of an EnvClass. This `envClass` object is the root of the class hierarchy tree whose `staticEnv` is the top-level environment.

A standard class (an instance of the Java `StdClass` class) has a similar constructor, except that it builds on the environment of its superclass as described earlier. A standard class defines bindings for the variables `myclass` and `superclass`, whose values are self-explanatory. You can find the code for this constructor in the file `class`.

Language OBJ (continued)

As described on slides 9 and 10, as an object is created, the `self` identifier is bound to the `objRef` reference that is passed to the `makeObject` method in `EnvClass`. But how can this binding take place when the object has not yet been completely created? We do this by creating a Java `ValRef` object named `objRef`, with a dummy initial value (`nil`, to be precise), and pass this to the `makeObject` method.

The `objRef` is passed up the superclass chain through successive `makeObject` calls. When `makeObject` reaches the `EnvClass` Java class, the `self` identifier is put into the object environment, bound to `objRef`. After the original object has been completely created – that is, after all of the `makeObject` calls have returned, `objRef` is rebound to the newly created object with a call to `setRef`. Here is the complete `eval` code for a `new` expression in the `NewExp` part of the `code` file:

```
public Val eval(Env env) {
    // get the class from which this object is created
    Val val = exp.eval(env);
    // create a reference to a dummy value (nil)
    Ref objRef = new ValRef(Val.nil);
    // let the class create the object
    ObjectVal objVal = val.makeObject(objRef);
    // set the reference to the newly created object
    return objRef.setRef(objVal);
}
```

Language OBJ (continued)

The `makeObject` method is defined for both a `StdClass` and an `EnvClass`. (For all other `Val` objects, `makeObject` throws an exception.)

In `StdClass`, `makeObject` first creates an instance of the superclass, and then stitches together an environment that includes the static bindings, the fields, and the methods. Three fields are created and initialized automatically: `self` is bound to the base object (a deep binding), `super` is bound to the instance of the superclass, and `this` is bound to this object (a shallow binding). The remaining named fields are bound to nil, and the methods are bound to closures as in `letrec`. The code for `makeObject` is given on the next two slides.

Language OBJ (continued)


```
public ObjectVal makeObject(Ref objRef) {
    // create the parent object first (recursively)
    ObjectVal parent = superClass.makeObject(objRef);

    // this object's environment extends the parent's environment
    Env env = parent.objectEnv;

    // add this class's static bindings
    env = env.extendEnvRef(staticBindings);

    // the fields come next
    Bindings fieldBindings = new Bindings();
    env = env.extendEnvRef(fieldBindings);
    // bind all of this object's instance fields to nil
    fields.addFieldBindings(fieldBindings);

    // bind all this object's methods as in letrec
    env = methods.addMethodBindings(env);
```

... continued on next slide ...

... continued from previous slide ...

```
      // create the object
      ObjectVal objectVal = new ObjectVal(env);

      // bind 'super' field to the parent object
      fieldBindings.add("super", new ValRef(parent)); // parent object
      // bind 'self' field to the base object being created
      // (to speed up lookups)
      fieldBindings.add("self", objRef); // deep
      // bind 'this' field to this object environment
      fieldBindings.add("this", new ValRef(objectVal)); // shallow
      return objectVal;
   }
```

Observe that this code binds `self` to `objRef` in every set of field bindings created recursively up to the top-level `EnvClass` class. This is not necessary, since the top-level `EnvClass` object is guaranteed to have a field binding of `self` to `objRef`, so any reference to `self` will eventually be found in the chain of environments. However, by putting the binding in every intermediate object, any reference to `self` will be found sooner in `applyEnv` lookups.

Language OBJ (continued)

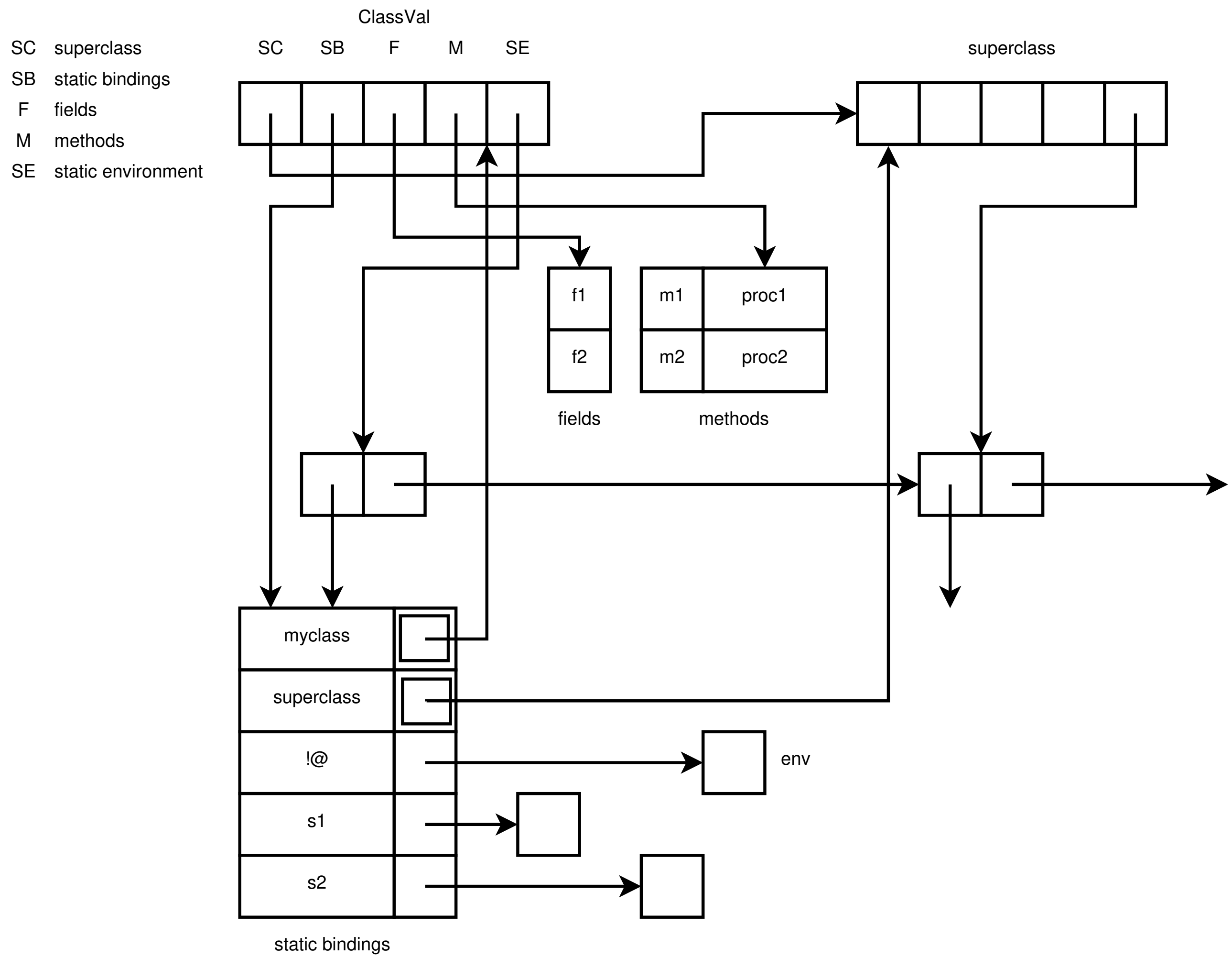
The `makeObject` method for `EnvClass` is simple, since it's always the last superclass object that needs to be constructed. It extends the EnvClass environment, namely the top-level environment, with a single field binding of `self` to (a reference to) the object being created. This binding is "deep", in the sense that the `objRef` reference may ultimately refer to an object defined in a deeply nested subclass. As shown in the `NewExp` code for `eval` on Slide 5.13, once the objects are created in the chain of superclasses, `objRef` is finally bound to to the object at the beginning of the chain.

```
public ObjectVal makeObject(Ref objRef) {
    // start with the static (top-level) env. of this class
    Env env = staticEnv;
    // add the field binding 'self' to refer to
    // the object being created (objRef)
    Bindings fieldBindings = new Bindings();
    fieldBindings.add("self", objRef);
    env = env.extendEnvRef(fieldBindings);
    return new ObjectVal(env);
}
```

Observe that an `ObjectVal` can access the static environment of the top-level class, which is the top-level program environment.

Language OBJ (continued)

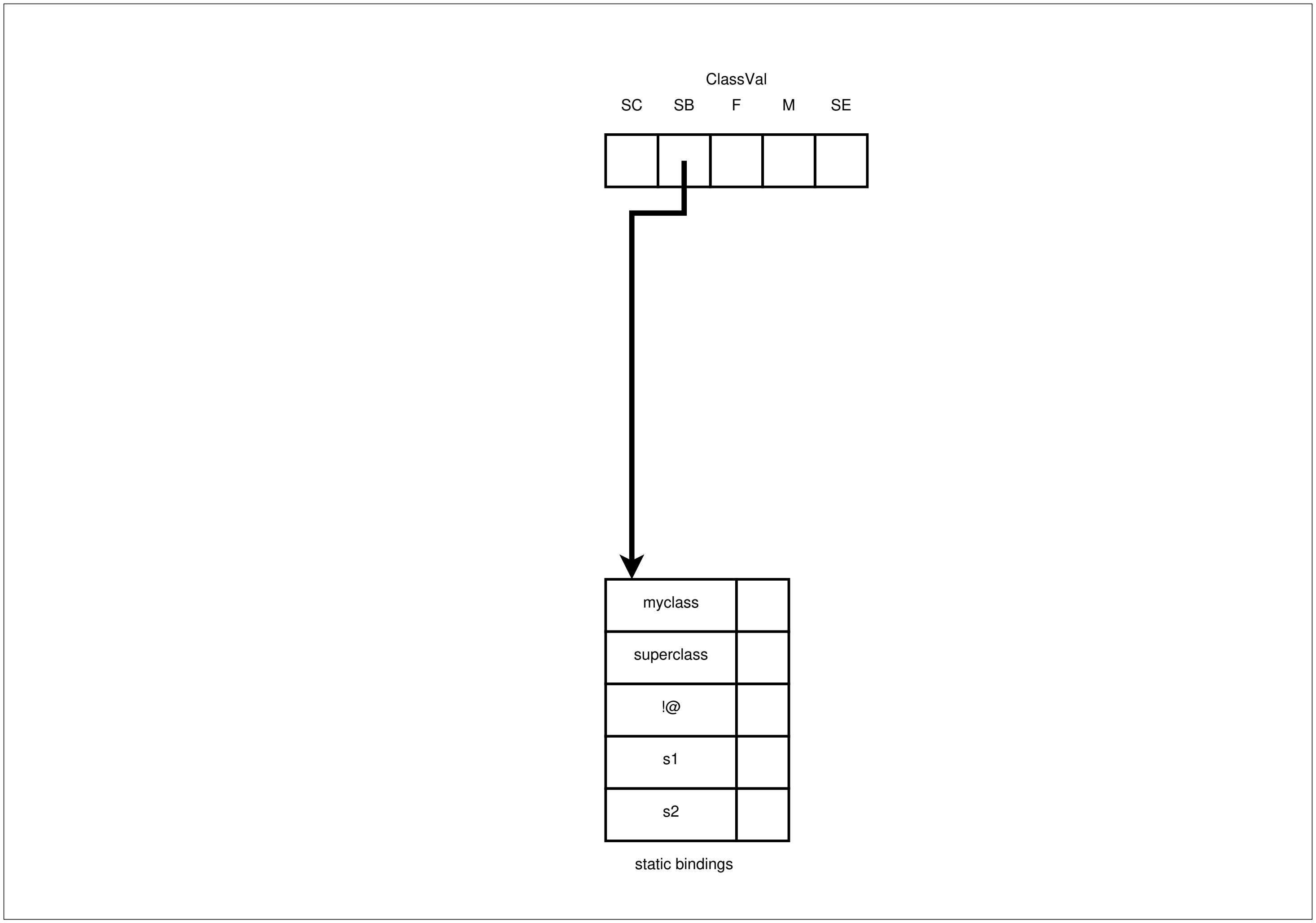
Here is a diagram showing the components of a `ClassVal`:



On the next six slides, we show step-by-step how to create an instance of an object using the `new` operator.

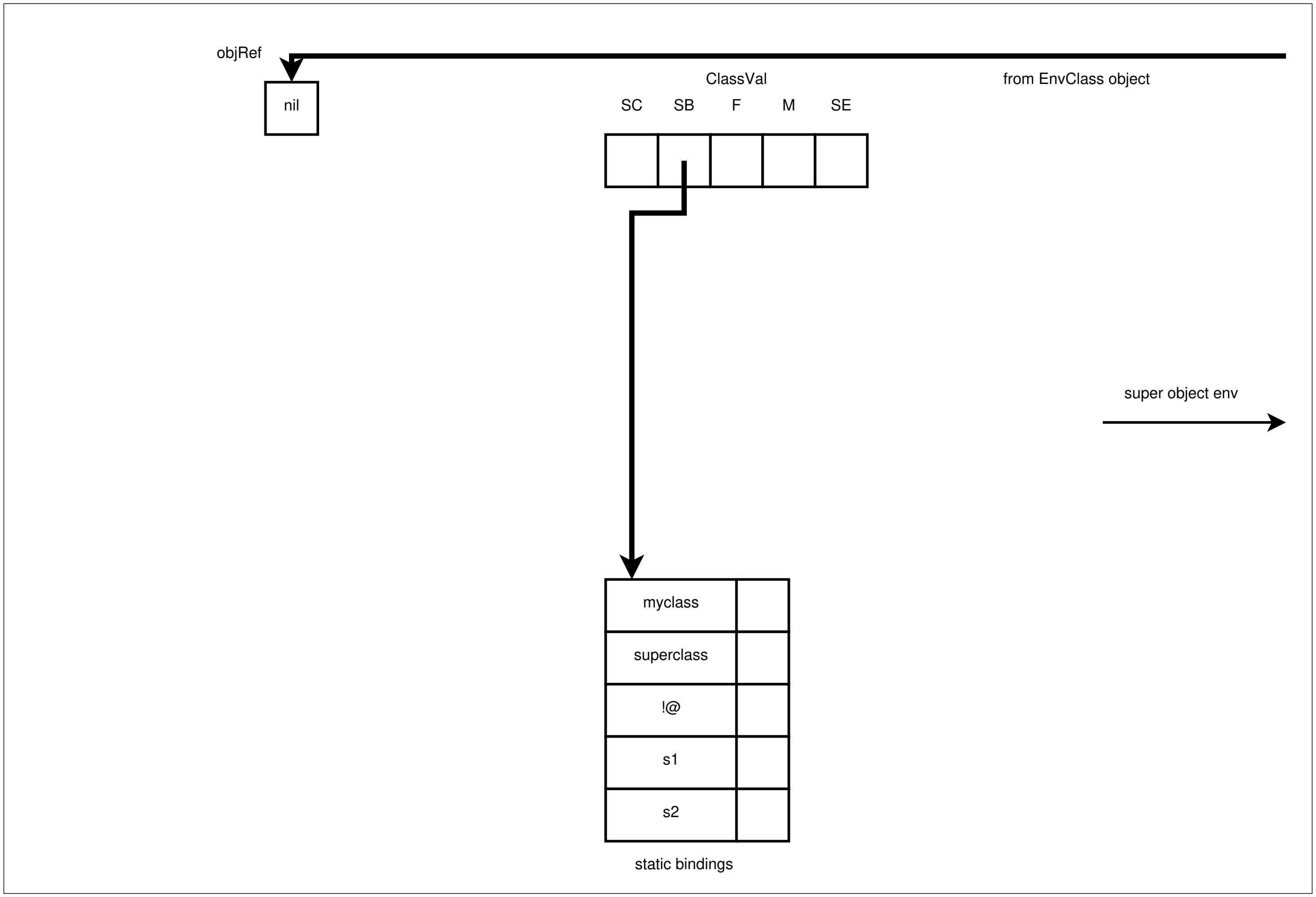
Language OBJ (continued)

Here is the class we want to create an instance of:



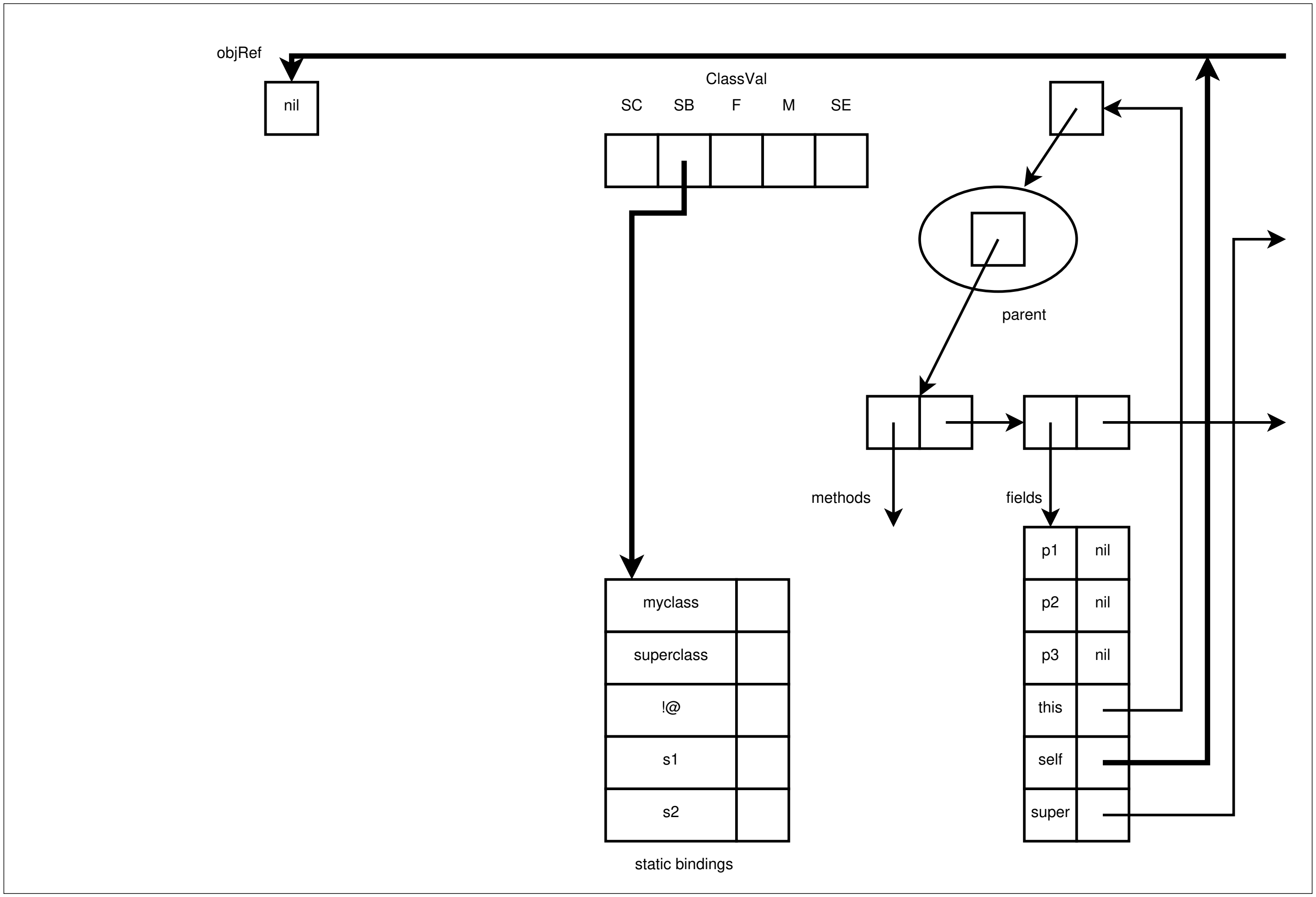
Language OBJ (continued)

Step 1: create a dummy reference `objRef` to a `NilVal`



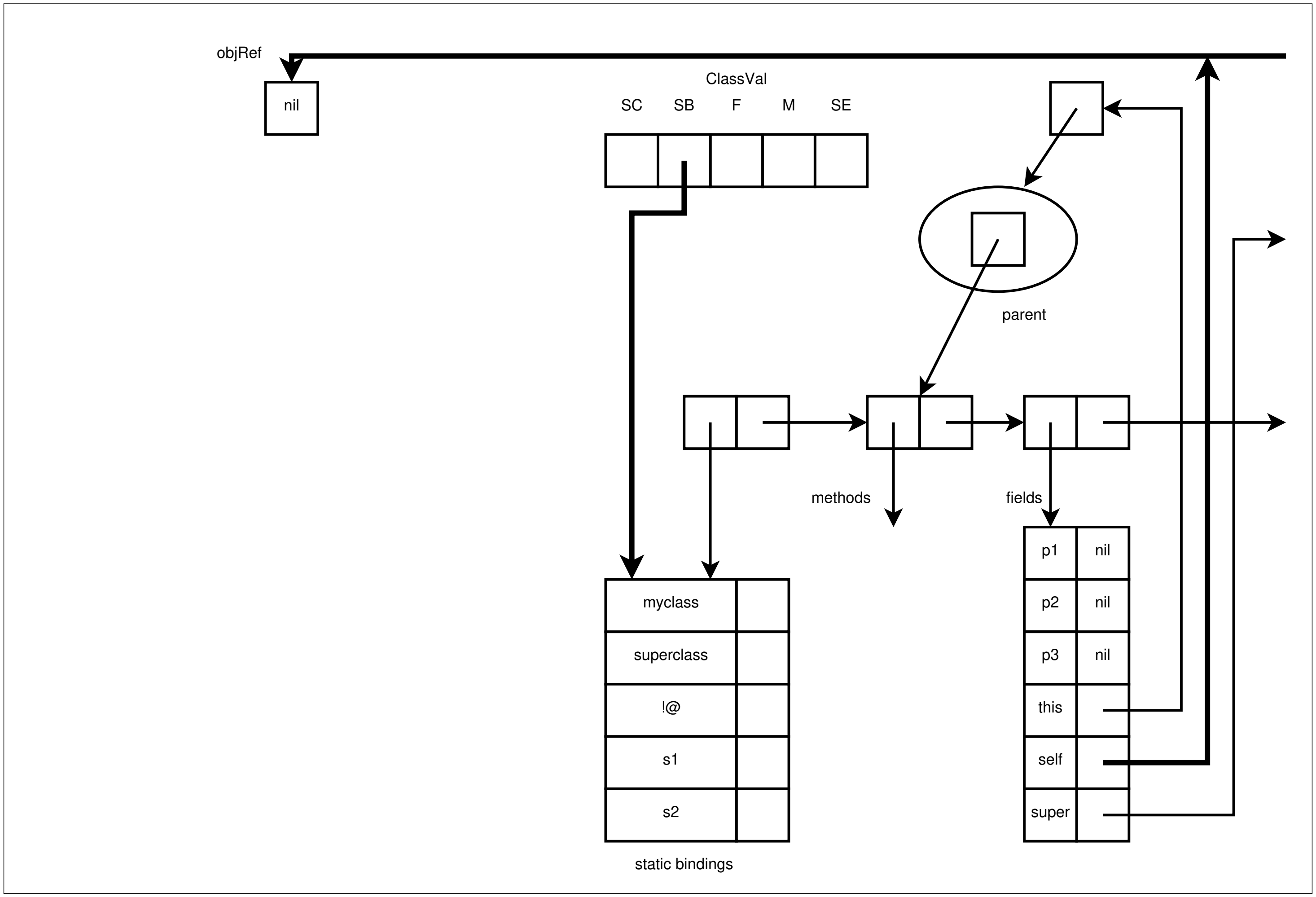
Language OBJ (continued)

Step 2: make a parent object (recursively), binding `self` to the object reference:



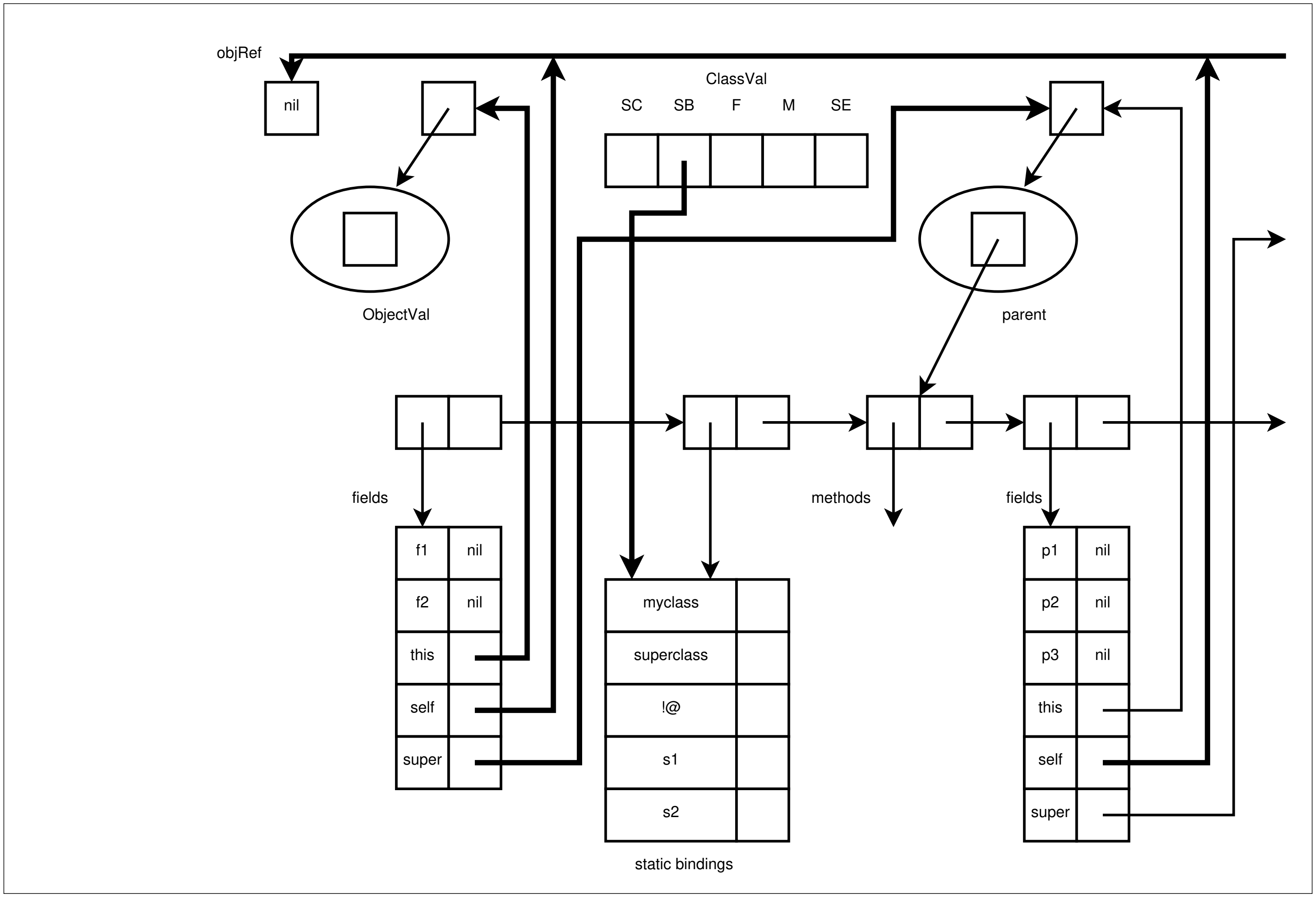
Language OBJ (continued)

Step 3: Extend the parent object environment by the static bindings of the class:



Language OBJ (continued)

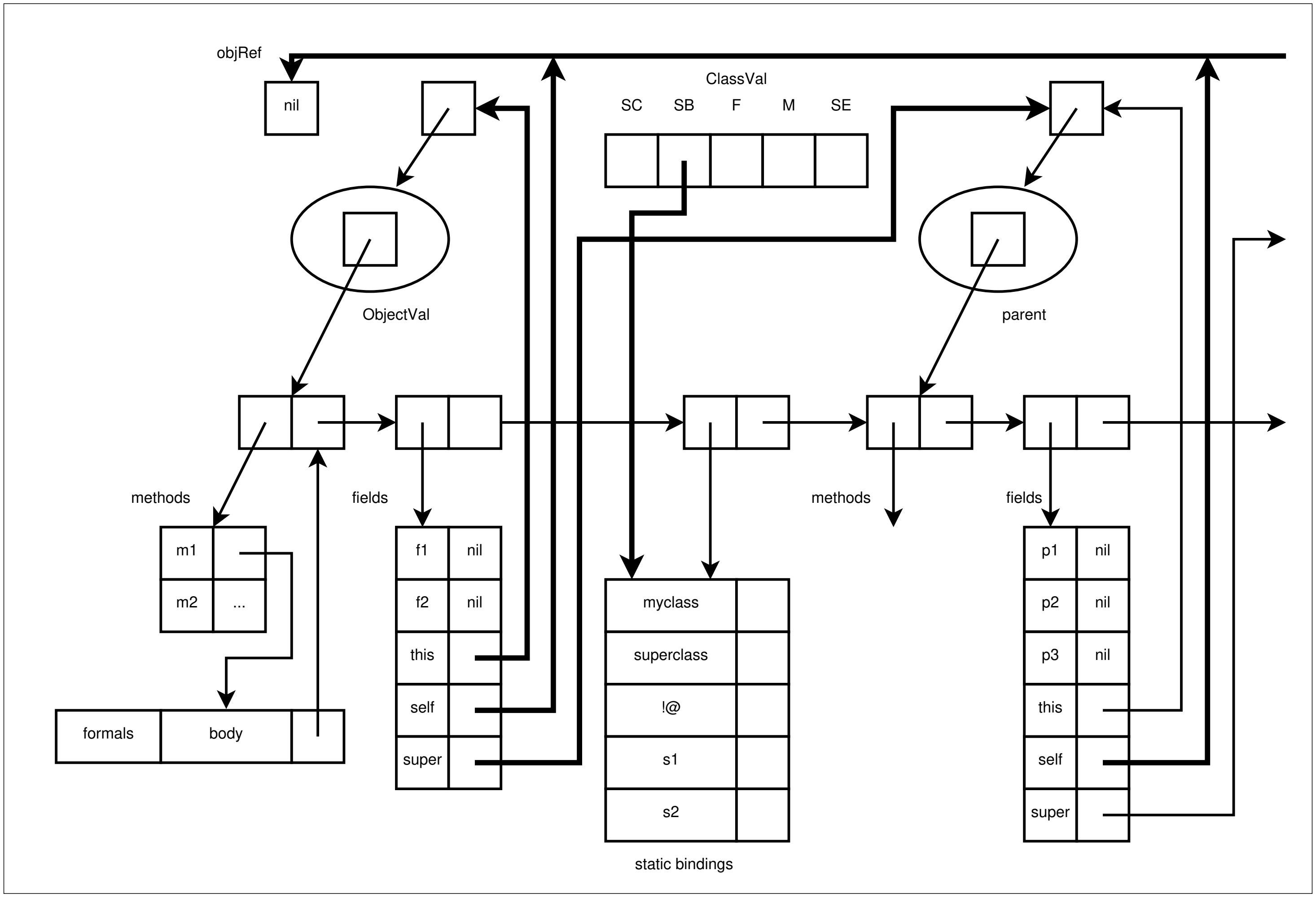
Step 4: Extend the environment of Step 3 with new field bindings



Fields are initialized to `nil`. Add bindings for `this`, `self`, and `super`.

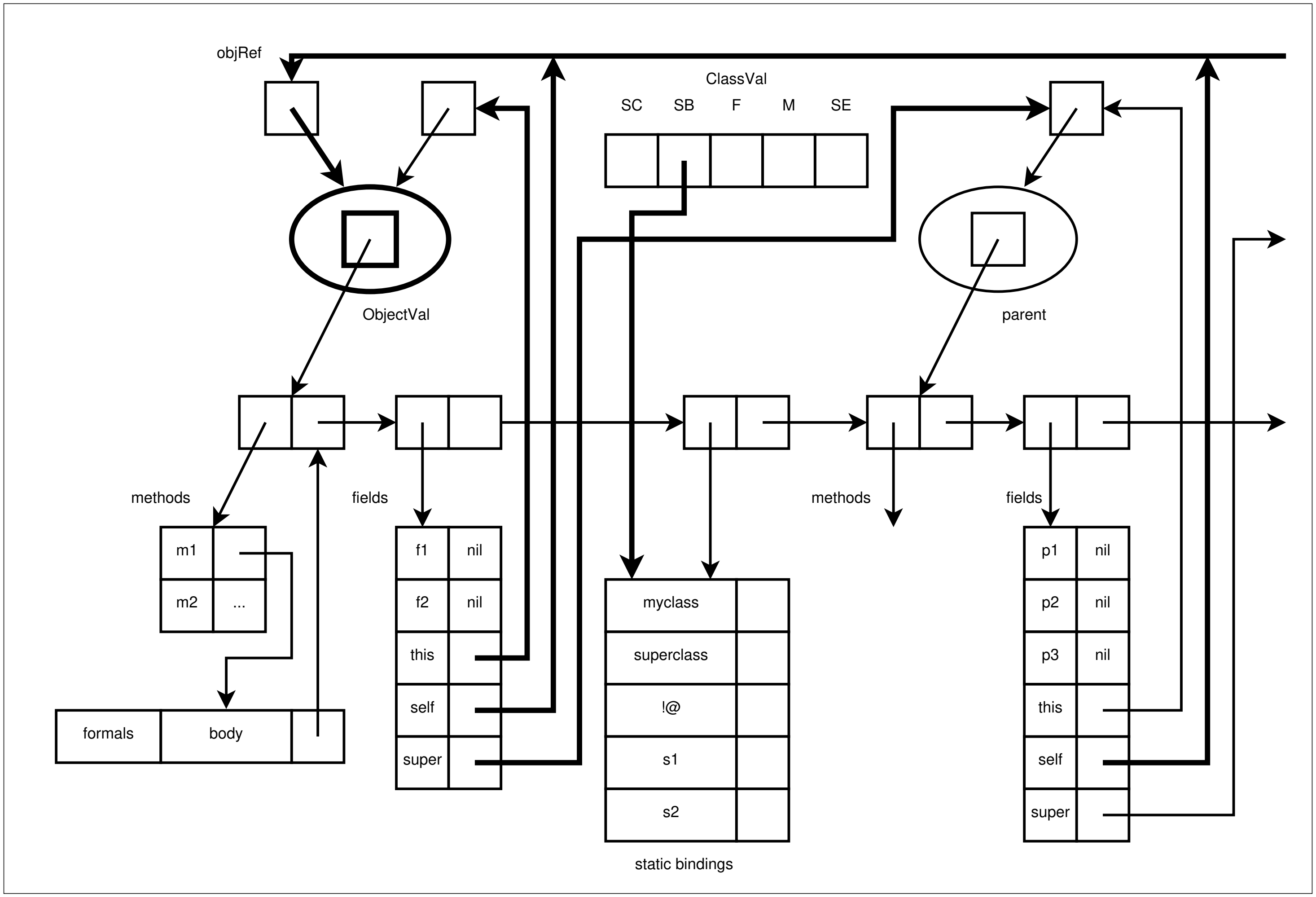
Language OBJ (continued)

Step 5: Extend the object environment with method bindings, as in `letrec`



Language OBJ (continued)

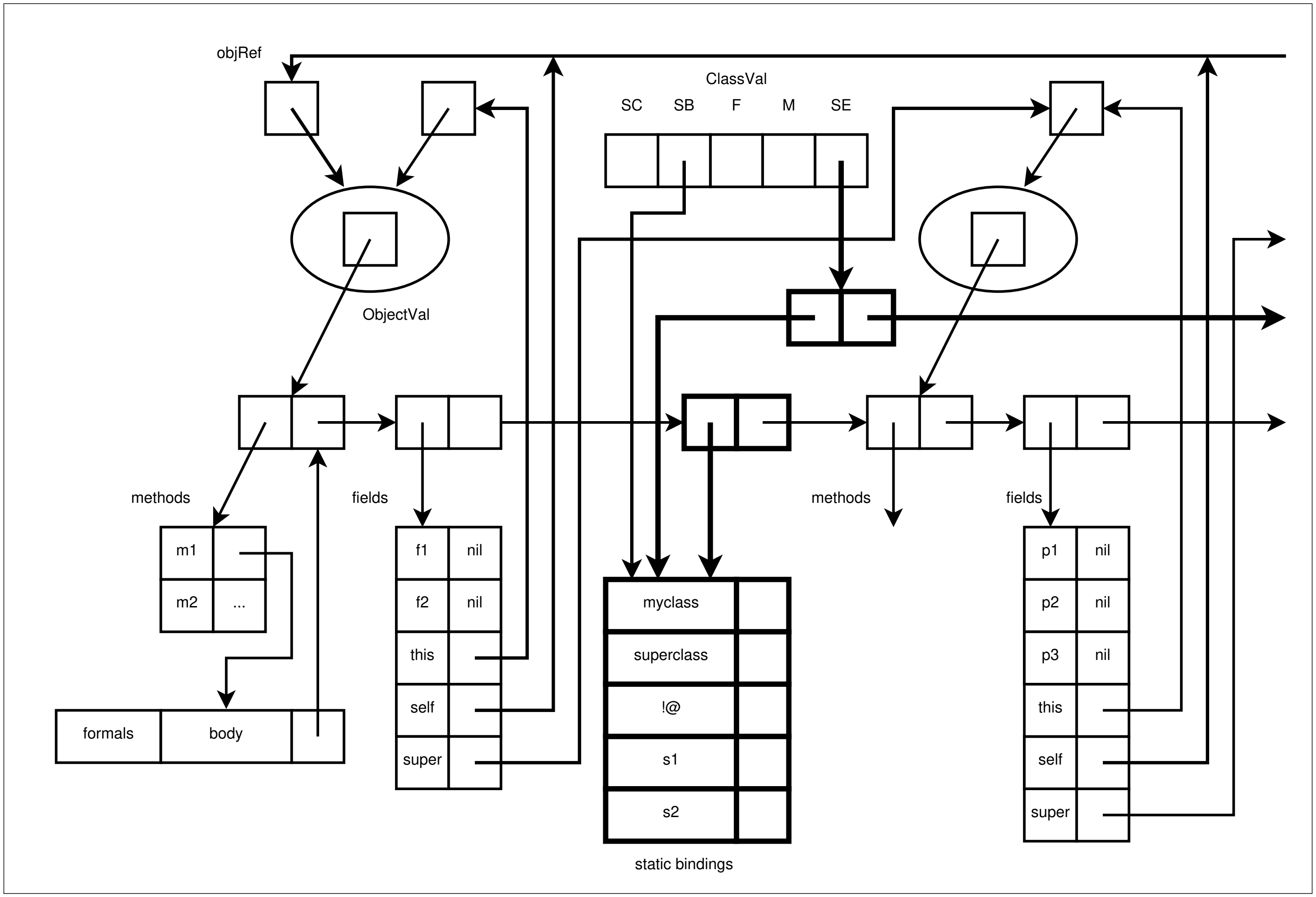
Step 6: Set `objRef` to refer to the newly created object



The `ObjectVal` instance is the value returned by the `new` operator.

Language OBJ (continued)

This diagram shows both the object environments and the static environments.



Observe that the class static bindings belong to both environments.

Language OBJ (continued)

The RHS expressions in static definitions are evaluated in the static environment of the class, which extends the static environment of its superclass (and so on...). For example, consider the following definitions and expression:

```
define c = class static x=5 end
define x = 3
<class extends c static y=x end>y
```

In the class that extends `c`, the value of `x` on the RHS of the `static y=x` expression can be found in the static environment of the superclass `c`, where `x` is bound to 5. Thus the variable `y` is bound to 5, not 3, so the expression evaluates to 5.

Now consider a variant of the expression:

```
define c = class static xx=5 end
define x = 3
<class extends c static y=x end>y
```

In the class that extends `c`, the value of `x` on the RHS of the `static y=x` expression can be found in the static environment of superclass `c`. But `c` does not define a variable `x`, so the value of `x` must be found in the static environment of *its* superclass, the singleton `EnvClass` that captures the top-level environment. Since `x` in the top-level environment is bound to 3, the variable `y` is bound to 3 as well, so this expression evaluates to 3.

Language OBJ (continued)

The <...> operator, when applied to a class, extracts the static environment of the class. When used in an expression of the form

```
<...>exp
```

the expression exp is evaluated in the static environment of the given class. This can be used to evaluate an expression that refers to any static variable in the class.

Consider a class c, for example. For a static variable x in the class c, the expression <c>x evaluates to the value of the variable. For a static procedure f in the class c, the expression .<c>f(...) evaluates to the application of f to its actual parameters.

When making a procedure application such as .<c>f(...), it is important to know

- the environment in which f is evaluated, and
- the environment in which the actual parameters to f are evaluated.

Language OBJ (continued)

To evaluate the expression `.<c>f(...)`, we evaluate the procedure `f` in the environment determined by `<c>`, whereas we evaluate the the actual parameter expressions in the environment in which the application `.<c>f(...)` is made. For example, in

```
define f = proc(t) *(2,t)
define c =
  class
    static x = 3
    static f = proc(t) t
  end
let
  x = 5
in
  .<c>f(x)
```

the expression `.<c>f(x)` evaluates to 5, because `f` is bound to the procedure given in the static definition of `c`, but its actual parameter `x` evaluates to 5, since the actual parameter expression is evaluated in the local `let` environment. **In other words, `.<c>f(x)` is the same as `.{<c>f}(x)`.**

Language OBJ (continued)

In particular, observe that the following two expressions are not equivalent:

```
.<c>f(...)
<c>.f(...)
```

In the first expression, `f` is evaluated in the static environment of `c`. Then the procedure bound to `f` is applied to the actual parameters `(...)` which are evaluated in the current environment, *not* in the static environment of `c`.

In the second expression, the entire expression `.f(...)` is evaluated in the static environment of `c`, which means that the actual parameters `(...)` are also evaluated in this static environment.

In the example on the previous slide, if the final expression was `<c>.f(x)` instead of `.<c>f(x)`, it would evaluate to 3, since `x` is bound to 3 in the static environment of `c`.

To clarify, the above two expressions can be re-written as follows:

```
.{<c>f}(...)
<c>{.f(...)}
```

The static environment of a class ultimately ends up extending the top-level program environment, not the environment in which the class is defined. Consider the following code:

```
define x = 3
let
  x = 5
in
  <class end>x
```

In this example, the class is defined in the `let` environment, but its static environment extends the top-level environment, not the `let` environment, so the value of this expression is 3, not 5.

There are situations in which we may want to retrieve the value of a variable in the "local" environment in which the class is defined and not in the static environment of the class. To do so, we predefine a static "variable" `!@` (called "bang-at") in every class and bind it to an object that captures the (local) environment in which the class is defined. This binding becomes part of the static environment of the class. The token '`!@`' is not really a variable, so it cannot appear in the LHS of an assignment. Its principal use is in expressions of the form

```
<!@>exp
```

which evaluates to the value of the expression `exp` in the local environment.

Language OBJ (continued)

For example, consider again the expression

```
define x = 3
let
  x = 5
in
  <class end>x
```

where we observed that this expression evaluates to 3. If we replace

```
<class end>x
```

in the above expression with

```
<class end><!@>x
```

then this expression evaluates to 5 because the `class` expression is defined in the `let` environment, and so the environment captured by `!@` has `x` bound to 5. The above expression can also be written as follows:

```
<<class end>!@>x
```

Observe that the "variable" `!@` is only defined in the static environment of a class and is bound to an object that captures the environment in which the class is defined. If used anywhere else, it will throw an "unbound variable" exception.

Language OBJ (continued)

The special operator `@` returns an object that captures the *current* environment, whatever that may be. (Recall your homework assignment that introduced this notation.) This operator may be used to pass a captured environment as an argument to a procedure application or to assign it to a variable for later reference.

```
AtExp
%%%
    public Val eval(Env env) {
        return new ObjectVal(env);
    }

%%%
```

The special operator `@@` does the same thing, except that it also displays the current environment in a human-readable way.

Notice that `@` is meaningful in *any* expression context – because every expression is evaluated in *some* environment – but that `!@` is only meaningful in an expression that appears in the context of a class, and its value represents an environment entirely separate from the static environment of the class.

Language OBJ (continued)

Some examples:

```
define x = 11
define y = 42
define z = 666
define xyenv =
  let
    x = 3
    y = 5
  in
    @
<@>x     % => 11
<@>y     % => 42
<@>z     % => 666
<xyenv>x % => 3
<xyenv>y % => 5
<xyenv>z % => 666 (the 'let' extends the top-level env)
```

Language OBJ (continued)

Observe that for any expression `exp`, the following two expressions evaluate to the same values:

```
<@>exp
exp
```

Language OBJ (continued)

Unlike the `new` operator in Java, Our `new` operator does not take any arguments, and all of the fields are initialized to `nil`. We can, of course, initialize fields by calling a method. Here's an example:

```
let
  c = class
        field x
        field y
        method init = proc(a,b) { set x=a ; set y=b ; self }
      end
  in
    let
      o = .<new c>init(3,4)
    in
      <o>+(x,y) % => 7
```

Since the `init` method returns `self`, the value of the expression `.<new c>init(3,4)` is the *same* object as the one created by the expression `new c`, except that its fields `x` and `y` are set to values 3 and 4, respectively.

You might be inclined to think that `.<new c>init(a,b)` is the same as `<new c>{ set x=a ; set y=b ; self}`, but the bindings for a and b may be different in these two expresions.

Language OBJ (continued)

In the previous example, the `init` method is invoked separately, after the object is created using `new`, and not as part of the object creation itself. Naming this procedure "`init`" is not a requirement. A class can have several methods that initialize its fields, much as a `Java` class can have several constructors. Unlike `Java`, the `OBJ` language can apply its methods – even the ones intended to initialize the fields – at any time.

```
define c =
    class
        field x
        method init = proc() {set x = 5 ; self}
        method foo = proc() {set x = add1(x) ; self}
    end

define o = .<new c>init()
<o>x                 % => 5
<o>{.foo() ; x}      % => 6
<o>{.init() ; x } % => 5
```

Language OBJ (continued)

The `OBJ` language has three additional expressions, with the following grammar rules:

```
<exp>:DisplayExp  ::= DISPLAY <exp>
                      DisplayExp(Exp exp)
<exp>:Display1Exp ::= DISPLAY1 <exp>
                      Display1Exp(Exp exp)
<exp>:NewlineExp  ::= NEWLINE
                      NewlineExp()
```

The `DISPLAY`, `DISPLAY1`, and `NEWLINE` tokens are defined by

```
DISPLAY 'display'
DISPLAY1 'display#'
NEWLINE 'newline'
```

Evaluating a `DisplayExp` expression results in evaluation of its `<exp>` part in the current environment; this value's `toString()` representation is then displayed on standard output, and the value is returned as the value of the `DisplayExp` expression. `Display1Exp` is like `DisplayExp` except that the displayed value is followed by a single space. The value of a `NewlineExp` expression is `nil`, and a newline is displayed on standard output.

Language OBJ (continued)

Here are examples of how to use `display`, `display#`, and `newline`:

```
let
  x = 3
  y = 5
  z = 8
in
  { display x ; newline
  ; display y ; newline
  ; display z ; newline
  ; nil
  }
```

Evaluating this expression results in displaying the following to standard output (omitting the final `nil`):

```
3
5
8
```

If the `newline` expressions are removed, the output appears as follows (omitting the final `nil`):

```
358
```

If `display` is then replaced by `display#`, the output appears as follows:

```
3 5 8
```

Language OBJ (continued)

Consider the following `OBJ` program:

```
define summer =
  class
    field sum
    method init = proc() {set sum = 0 ; self}
    method add = proc(t) {set sum = +(sum,t) ; self}
    method show = proc() {display sum ; newline ; self}
  end
```

Here's an example of how this class might be used to find and display the sum of the integers 1, 3, 5, and 7:

```
define o = .<new summer>init()
.<o>add(1)
.<o>add(3)
.<o>add(5)
.<o>add(7)
.<o>show()
```

Since `init` and `add` both return `self`, the following "one-liner" would accomplish the same thing:

```
.<.<.<.<.<.<new summer>init()>add(1)>add(3)>add(5)>add(7)>show()
```

Language OBJ (continued)

Since we often find ourselves encountering expressions like the following:

```
.<.<.<.<.<.<new summer>init()>add(1)>add(3)>add(5)>add(7)>show()
```

we introduce a short-hand way of writing this:

```
!<new summer>init()>add(1)>add(3)>add(5)>add(7)>show()!>
```

The token `LLANGLE`, defined as the string '`!<`', introduces an expression – which must evaluate to an environment (class or object) – followed by a sequence of zero or more procedure applications, each preceded by '>'. Each procedure is evaluated in the environment of the previous class or object, and the procedure application itself must return another object, whose environment is then used to evaluate the next procedure, and so on. The entire expression is terminated by a `RRANGLE` token, defined as '`!>`'. Note that the actual parameter expressions are evaluated in the environment in which the entire expression appears.

Language OBJ (continued)

Here are the associated grammar rules:

```
<exp>:EenvExp ::= LLANGLE <exp> <mangle> RRANGLE
                  EenvExp(Exp exp, Mangle mangle)
<mangle>        **= RANGLE <exp> LPAREN <rands> RPAREN
                  Mangle(List<Exp> expList, List<Rands> randsList)
```

The BNF identifier `mangle` should suggest “multiple angle (brackets)”, but it also could also appropriately be interpreted as a twisted mess.

Language OBJ (continued)


```
<exp>:EenvExp   ::= LLANGLE <exp> <mangle> RRANGLE
                    EenvExp(Exp exp, Mangle mangle)
<mangle>:Mangle **= RANGLE <exp> LPAREN <rands> RPAREN
                    Mangle(List<Exp> expList, List<Rands> randsList)
```

Here is the implementation of how to evaluate an `EenvExp` expression.

```
EenvExp
%%%
    public Val eval(Env env) {
        Val v = exp.eval(env);      // the environment object
        return mangle.eval(v, env);
    }
%%%
```

The expression `exp` is evaluated in the current environment. Its value `v`, along with the current environment, is passed to the `mangle` object, which then evaluates the subsequent procedure applications.

Language OBJ (continued)


```
<exp>:EenvExp    ::= LLANGLE <exp> <mangle> RRANGLE
                     EenvExp(Exp exp, Mangle mangle)
<mangle>:Mangle  **= RANGLE <exp> LPAREN <rands> RPAREN
                     Mangle(List<Exp> expList, List<Rands> randsList)
```

A `mangle` object consists of a list of `Exp` objects and a list of `Rands` objects.

Language OBJ (continued)

Here is the code for `eval(Val v, Env env)` in the `Mangle` class:

```
public Val eval(Val v, Env env) {
    Iterator<Exp> expIter = expList.iterator();
    Iterator<Rands> randsIter = randsList.iterator();
    while (expIter.hasNext()) {
        // expIter.next() ProcExp to apply
        // v.env() is the environment in which to build the ProcVal
        Val p = expIter.next().eval(v.env());
        // evaluate this method's rands in env
        List<Val> valList = randsIter.next().evalRands(env);
        v = p.apply(valList);
    }
    return v;
}
```

Each `Exp` is evaluated in the environment defined by the value `v` (which must, perforce, be a class or object), and this must evaluate to a `ProcVal` (named `p` in this code). The `valList` arguments to `p` are evaluated in the outside environment `env` (*not* in the environment defined by `v`) from the corresponding `Rands` object. The procedure `p` is then applied to these arguments, and the resulting application becomes the new `v`. This is repeated until all of the `Mangle` applications are performed. The final value `v` is returned as the result of the `EenvExp` expression.

Language OBJ (continued)

It turns out that exposing the environment of a procedure can be used to implement, with procedures alone, a simplified approach to objects and methods. See the file

```
OBJ/Prog/pobj
```

for an example. On the other hand, exposing the environment of a procedure can result in modifying that environment, which can lead to unintended consequences.

Language OBJ (continued)

In method application, `self` always refers to the base object, even if `self` appears in the definition of a superclass method. This is what makes dynamic dispatch work!

```
define c1 =
  class
    method m1 = proc() 1
    method m2 = proc() <self>m1()
  end
define c2 =
  class extends c1
    method m1 = proc() 2
  end
define o1 = new c1
define o2 = new c2

.<o1>m1() % => 1
.<o2>m1() % => 2
.<o2>m2() % => 2!
```

The following slide gives another example of dynamic dispatch.

Language OBJ (continued)


```
define shape =
  class
    method area = proc() 0 % shapeless
  end

define rectangle =
  class extends shape
    field len % length
    field wid % width
    method init = proc(len,wid) {set <this>len=len ; set <this>wid=wid ; self}
    method area = proc() *(len,wid)
  end

define circle =
  class extends shape
    field rad % radius
    method init = proc(rad) {set <this>rad=rad ; self}
    method area = proc() *(3,*(rad,rad)) % a bit of an underestimate
  end

define r = .<new rectangle>init(4,5) % a rectangle with length 4 and width 5
define c = .<new circle>init(2)      % a circle with radius 2
define s = new shape

.<r>area() % => 20
.<c>area() % => 12
.<s>area() % => 0
```

Language OBJ (continued)

Other examples on this slide and the next ...

```
define c1 =
  class
    method m1 = proc() <self>m2()
    method m2 = proc() 13
  end
define c2 =
  class extends c1
    method m1 = proc() 22
    method m2 = proc() 23
    method m3 = proc() <super>m1()
  end
define c3 =
  class extends c2
    method m1 = proc() 32
    method m2 = proc() 33
  end
define o3 = new c3

<o3>m3()  % => 33
```

Language OBJ (continued)


```
define a = class
    field i field j
    method setup = proc() {set i=15; set j=20; 50}
    method f = proc() .<self>g()
    method g = proc() +(i,j)
  end
define b = class extends a
    field j field k
    method setup =
      proc() {set j=100; set k=200; .<super>setup(); .<self>h()}
    method g = proc() [i,j,k]
    method h = proc() .<super>g()
  end
define c = class extends b
    method g = proc() .<super>h()
    method h = proc() +(j,k)
  end
let
   p = proc(o)
     let
       u = .<o>setup()
     in
       [u, .<o>g(), .<o>f()]
in
  [.p(new a), .p(new b), .p(new c)]

% returns [[50,35,35],[35,[15,100,200],[15,100,200]],[300,35,35]]
```

Language PROP

In many object-oriented programming languages, the fields of an object can be made *private* – that is, inaccessible outside of the object's methods. For private fields, special publically accessible methods can be used to retrieve the field values or to modify them. These methods are often called *getters* and *setters*, respectively.

Suppose, for example, we provided a special designator called `private` that served to identify a field whose value was inaccessible outside of the class methods. Consider the following code:

```
define c =
  class
    private x
    method get_x = proc() x
    method set_x = proc(v) set x = v
  end
define cc = new c
.<cc>set_x(5) % ok - sets value of x to 5
.<cc>get_x()  % ok - returns 5
<cc>x            % illegal - x is private
```

While this sort of code is common, there are two problems with this approach. The first is that every private field we want to access needs a getter and a setter. The second is that code such as `<cc>set x = 5` does not work, but is easier to write and understand than `.<cc>set_x(5)`.

Language PROP (continued)

The `C#` language championed by Microsoft solves these problems using the notion of a *property*. A property acts like a field but it provides built-in getter and setter code. When the field is *accesssed*, the getter code is executed; when the field is *assigned to* with a `set` statement, the setter code is executed.

Here is the same class as described on the previous slide, with a property instead of a getter and setter:

```
define c =
  class
    field x
    property x = prop x:set x=$
  end
define cc = new c
<cc>set x = 5 % ok - sets the field value to 5
<cc>x         % returns 5
```

The property `x` shadows the field `x`. This means that the field `x` cannot be accessed directly except through the property.

Language PROP (continued)

Here are the grammar rules for defining properties in a class defintion:

```
<props> **= PROPERTY <VAR> EQUALS <prop>
        Props(List<Token> varList, List<Prop> propList)
<prop>  ::= PROP <exp>getExp COLON <exp>setExp
        Prop(Exp getExp, Exp setExp)
```

When a variable bound to a property is evaluated, its `getExp` code is executed using the environment captured where the property is defined, and the expressed value of the variable is the result of evaluating the `getExp` expression.

When a variable bound to a property is assigned to in a `set` expression, the RHS of the expression is evaluated in the current environment. The environment where the property is defined is then extended by binding the special symbol '`$`' to the value of the RHS. The property's `setExp` expression is then evaluated in this extended environment, and the resulting value is the value of the `setExp` expression.

Language PROP (continued)

If a variable `z` [for example] is bound to a property whose `set` expression has no side-effects [such as `nil`], an expression such as `set z = ...` does not modify anything – including `z`.

```
define c =
  class
    field x
    method init = proc(x) {set <this>x = x ; self}
    property y = prop x:set x=$
    property z = prop x:nil
  end
define o = .<new c>init(3)
<o>[x,y,z]   % [3,3,3]
<o>set x = 5
<o>[x,y,z]   % [5,5,5]
<o>set y = 11
<o>[x,y,z]   % [11,11,11]
<o>set z = 42
<o>[x,y,z]   % [11,11,11]
```

Language PROP (continued)

We implement properties in the same way we implement call-by-name: a property evalutes to a thunk-like object called a `PropRef` (which extends the `Ref` class); this object captures the environment in which the property is defined, along with the property's `get` and `set` expressions.

The expressed value of a variable bound to a property is the value obtained by evaluating the property's `get` expression in the captured environment by calling the thunk's `deRef` method.

Similarly, when assigning a value to a variable bound to a property, the thunk's `setRef` method is called, with the assigned value bound to the special variable '`$`' and returning the value of the property's `set` expression.

Language PROP (continued)

If a field named `x` in a class has a property also named `x`, referring to `x` in the contect of this object uses the `PropRef` instead of the variable. Consider this example:

```
define c =
  class
    static p = proc(t) set t=add1(t)
    field x
    method init = proc(x) {set <this>x=x ; self}
    property x = prop x : set x=$
    property y = prop x : set x=+($,$)
  end
define o = .<new c>init(3)
<o>{.p(x) ; x} % evaluates to 4
<o>{.p(y) ; x} % evaluates to 10
```

In the expression `.p(x)`, `x` refers to the property `x`. Since we are using call-by-reference parameter passing semantics, the variable `t` is bound to this property, and the expression `set t=add1(t)` is evaluated using this binding. Remember that the RHS parts of property definitions are all evaluated in the environment that includes only statics, fields, and methods, but not properties.

Language PROP (continued)

So far, a property is only defined in the conext of a class definition, where it plays a role in object instantiantion. It turns out that the behavior of properties could be useful even outside of the context of an object, especially to manage access to variables defined in a `let` expression. To make this explicit, we create a `letprop` construct that has the following concrete syntax and abstract class structure:

```
<exp>:LetpropExp  ::= LETPROP <letpropDecls> IN <exp>
                      LetpropExp(LetpropDecls letpropDecls, Exp exp)
<letpropDecls>     **= <VAR> EQUALS <prop>
                      LetpropDecls(List<Token> varList, List<Prop> propList)
```

Consider this (somewhat strange) example:

```
let
  x = 3
in
  letprop
    x = prop x : set x = 5
  in
    {set x = 42 ; x} % => 5
```

This expression evaluates to 5.

Language PROP (continued)

The `set` part of a `prop` is optional. If the `set` part is omitted, any attempt to apply the `set` operator to the variable results in a runtime exception. In this way, we can implement “read-only” properties.

```
let
  x = 3
in
  letprop
    x = prop x %% no 'set' part, so x is read only
  in
    {set x = 42 ; x} % => runtime exception
```

Language PROP (continued)

It turns out that a read-only `prop` behaves just like a thunk in call-by-name (which, you may recall, is also read-only), so we have the benefit of call-by-name semantics when we want it!

```
let
  while = proc(test?, do, ans)
    letrec
      loop = proc()
        if test? then {do ; .loop()} else ans
    in
      .loop()
  sum = 0
  count = 10
  i = 1
in
  letprop
    test? = prop count
    do = prop { set sum = +(sum, i)
              ; set i = add1(i)
              ; set count = sub1(count)
              }
    ans = prop sum
  in
    .while(test?, do, ans) % => 55 = 1+2+...+10
```