Classes and Objects

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

A *class* is an expressed value that captures a collection of variable
and procedures (called *methods*) and that serves as a template to c
values called *instances* of the class. The instances are called *objects*
them from classes. You can think of a class as a factory that identifie
out an arbitrary number of instances (objects) of the class.

Language OBJ

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

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

When an object of class X is instantiated, instances of each of the cla
the path from X to the root of the tree are created, and the combinat
instances is considered as the resulting object.

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

A class may also have `static` variables whose values are sharec
stances of the class.

If x is an object and f is a field, the expression `<x>f` evaluates to t
to f in object x. In languages like C++ and Java, this would be wr
`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
of c2 and `<o2>super` is an instance of c1.

Language OBJ (continued)

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

```
<exp>:NilExp    ::= NIL
                NilExp()
<exp>:ClassExp  ::= CLASS <ext> <statics> <fields> <methods> EN
                ClassExp(Ext ext, Statics statics, Fields fiel
<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> procLi
<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 `ClassEx`
tion, so evaluating such a `ClassExp` (*i.e.*, calling its `eval` method
object. Looking only at the syntax, it seems reasonable that such a
(possibly empty) superclass (the `ext` [for `EXTENDS`] part), a list o
fiers bound to values, a list of field names, and a list of method n
procedures.

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

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

Language OBJ (continued)

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

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

Language OBJ (continued)

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

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

The `staticEnv` variable is a reference to the static environmen
it starts out extending the static environment of the superclass
`Bindings` object named `staticBindings`. New bindings ar
`staticBindings` list using the variable definitions in the `sta`
eter of the `StdClass` constructor. Each static LHS identifier i
value of its RHS, where the RHS expression is evaluated in the curr
ronment. These bindings are created in order (first to last) as in top-l

Language OBJ (continued)

Two predefined identifiers are inserted initially into the list of static
binds the identifier `myclass` to (a reference to) this class itself, an
the identifier `superclass` to (a reference to) the superclass of thi

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

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

If a class expression has a static definition for a variable that also ap
variable in a superclass, that definition shadows the superclass vari
low duplicate LHS variable names in a given class expression's st
We also disallow static redefinitions for `myclass` or `superclas`

Language OBJ (continued)

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

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

```
public Env objectEnv;
```

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

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

The method variable names in the class definition are bound to pro
that capture the environment that includes bindings for the class
ables (from the `staticBindings` field), along with bindings f
described above. The method closures are created as in `letrec`, s
to themselves recursively. As with static and field definitions, we di
method variable names.

Language OBJ (continued)

Before we build an object from a base class, we first build an ol instance of the superclass of the base class. This superclass object ha ronment, namely `objectEnv`. We then extend this superclass obje by adding bindings to the statics, fields, and methods of the class, a tended environment to create the instance of the base class.

Since creating the superclass object may itself involve creating *its* superclass, object creation continues up the class hierarchy unt `EnvClass` class is found, at which point there is no further super create.

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

Language OBJ (continued)

As each object is created up the superclass chain, we insert three pr into its list of field bindings: `this`, `self`, and `super`. We bind the `self` to the base object being created, which is the same as descri bind the field identifier `this` to the object being created at the particu superclass chain (we call this a *shallow* binding). And we bind the `super` to the superclass object. The code for creating these bindin Slide 5.16.

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

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

Language OBJ (continued)

A an instance of the Java `EnvClass` class has one instance variabl

```
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 stance of an EnvClass. This `envClass` object is the root of the tree whose `staticEnv` is the top-level environment.

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

Language OBJ (continued)

As described on slides 9 and 10, as an object is created, the sel bound to the `objRef` reference that is passed to the `makeObje` `EnvClass`. But how can this binding take place when the object h completely created? We do this by creating a Java `ValRef` object na with a dummy initial value (`nil`, to be precise), and pass this to the method.

The `objRef` is passed up the superclass chain through successive calls. When `makeObject` reaches the `EnvClass` Java class, the s is put into the object environment, bound to `objRef`. After the orig been completely created – that is, after all of the `makeObject` calls `objRef` is rebound to the newly created object with a call to `setR` complete `eval` code for a `new` expression in the `NewExp` part of th

```
public Val eval(Env env) {
    // get the class from which this object is c
    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 ob
    return objRef.setRef(objVal);
}
```

Language OBJ (continued)

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

In `StdClass`, `makeObject` first creates an instance of the super
stitches together an environment that includes the static bindings, the
methods. Three fields are created and initialized automatically: `se`
the base object (a deep binding), `super` is bound to the instance of
and `this` is bound to this object (a shallow binding). The remainin
are bound to nil, and the methods are bound to closures as in `let`
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'

    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
    fields.addFieldBindings(fieldBindings);

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

... continued on next slide ...

Language OBJ (continued)

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

Observe that this code binds `self` to `objRef` in every set of fiel ated recursively up to the top-level `EnvClass` class. This is not n the top-level `EnvClass` object is guaranteed to have a field bindi `objRef`, so any reference to `self` will eventually be found in the ch ments. However, by putting the binding in every intermediate object to `self` will be found sooner in `applyEnv` lookups.

Language OBJ (continued)

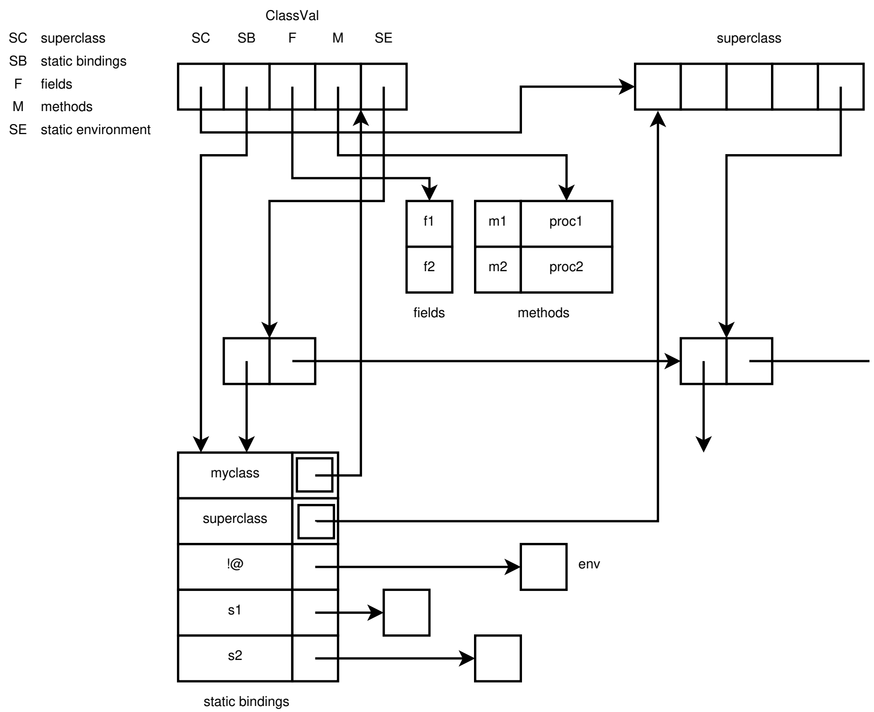
The `makeObject` method for `EnvClass` is simple, since it's alw perclass object that needs to be constructed. It extends the EnvClas namely the top-level environment, with a single field binding of s erence to) the object being created. This binding is "deep", in the `objRef` reference may ultimately refer to an object defined in a subclass. As shown in the `NewExp` code for `eval` on Slide 5.13, o are created in the chain of superclasses, `objRef` is finally bound to the beginning of the chain.

```
    public ObjectVal makeObject(Ref objRef) {
        // start with the static (top-level) env. of thi
        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 c class, which is the top-level program environment.

Language OBJ (continued)

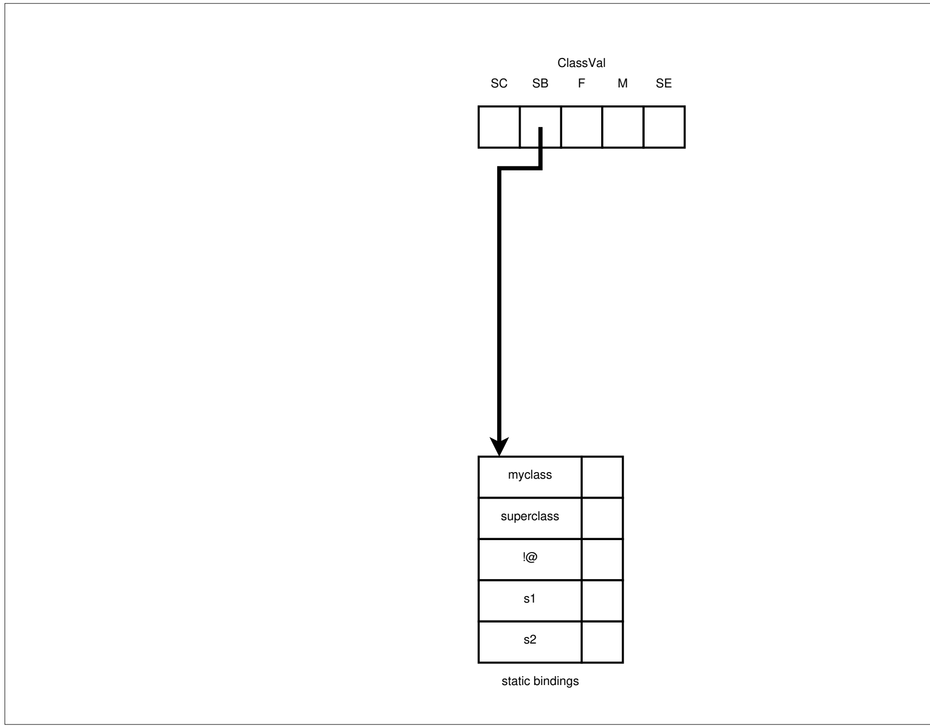
Here is a diagram showing the components of a `ClassVal`:



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

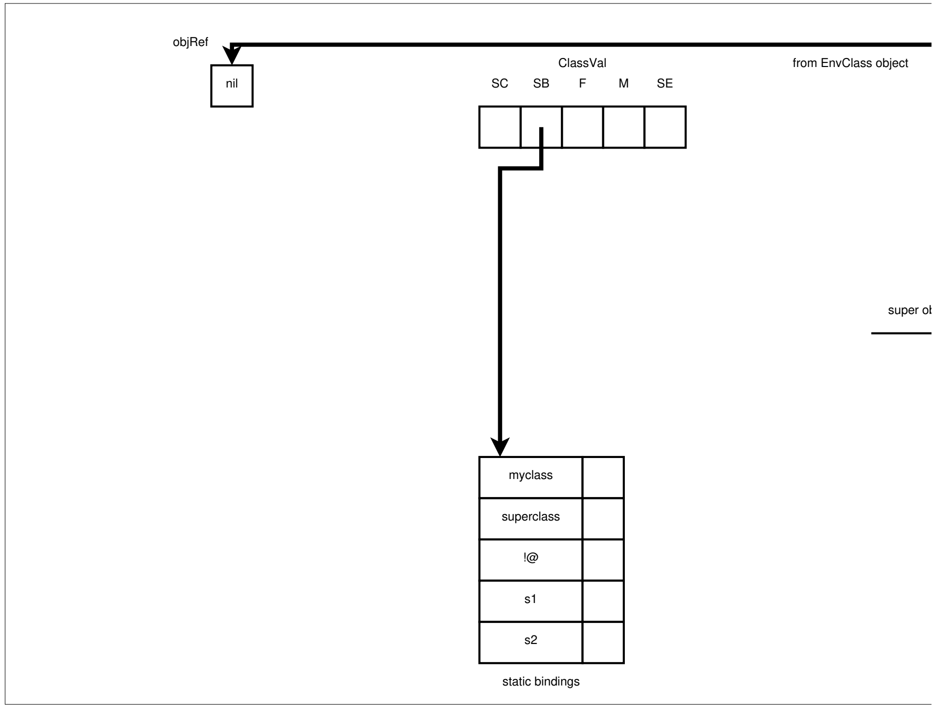
Language OBJ (continued)

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



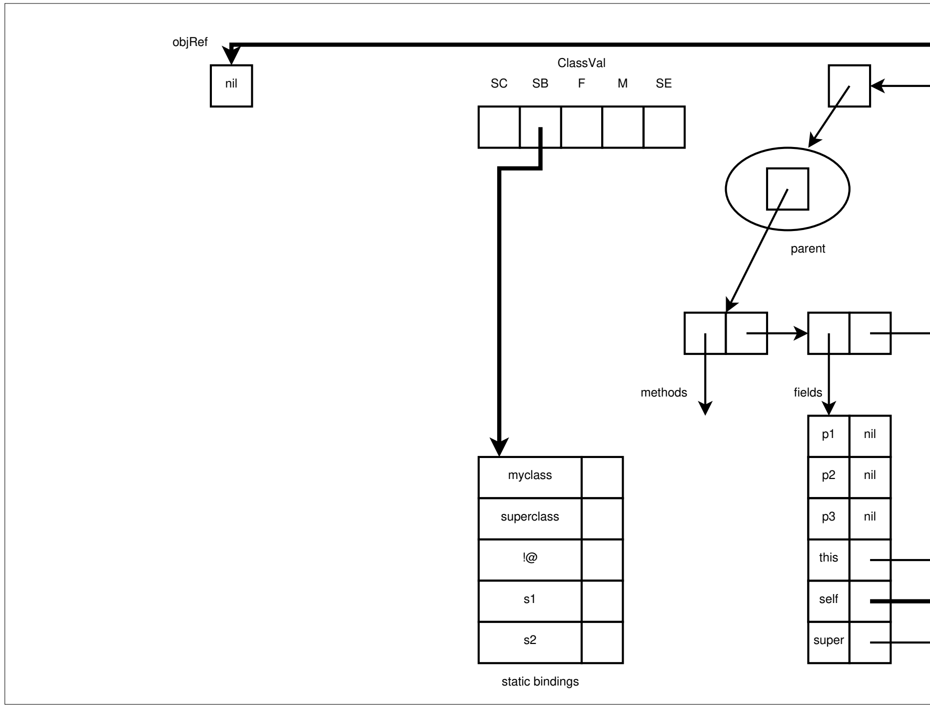
Language OBJ (continued)

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



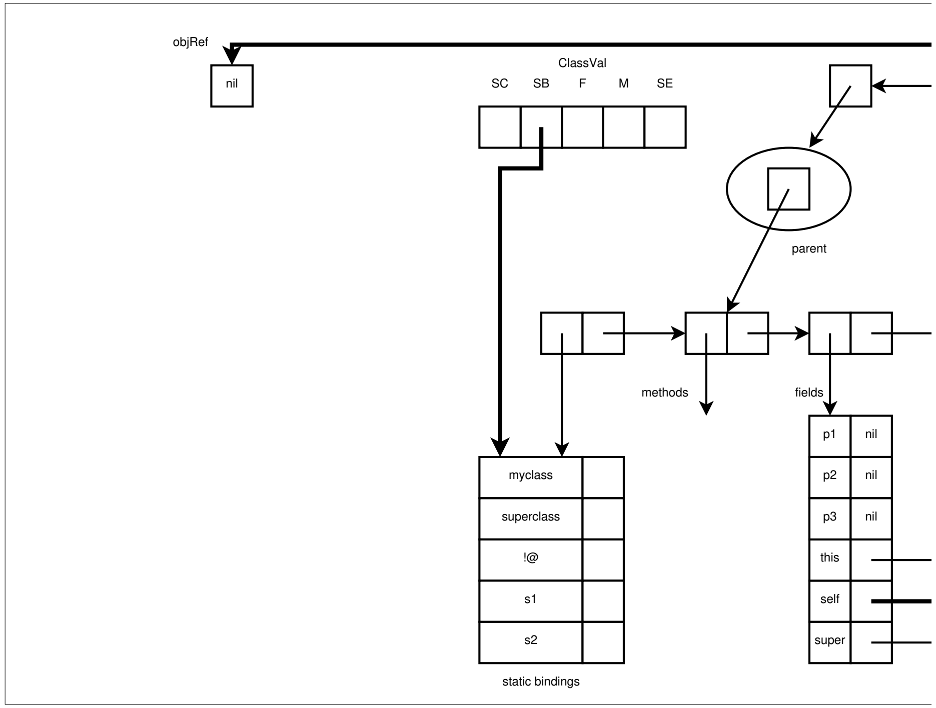
Language OBJ (continued)

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



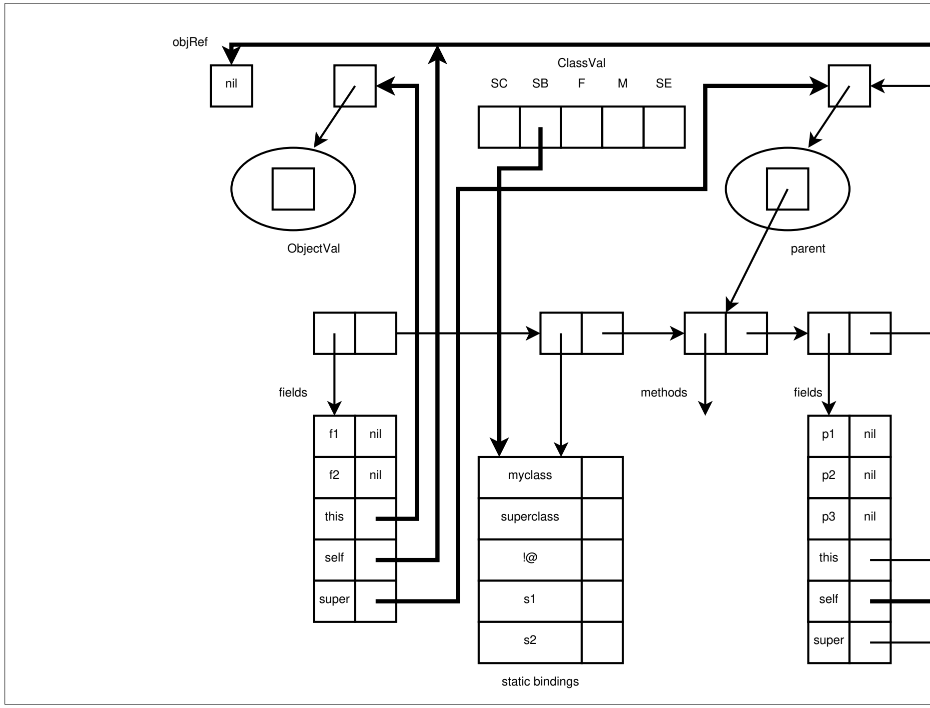
Language OBJ (continued)

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



Language OBJ (continued)

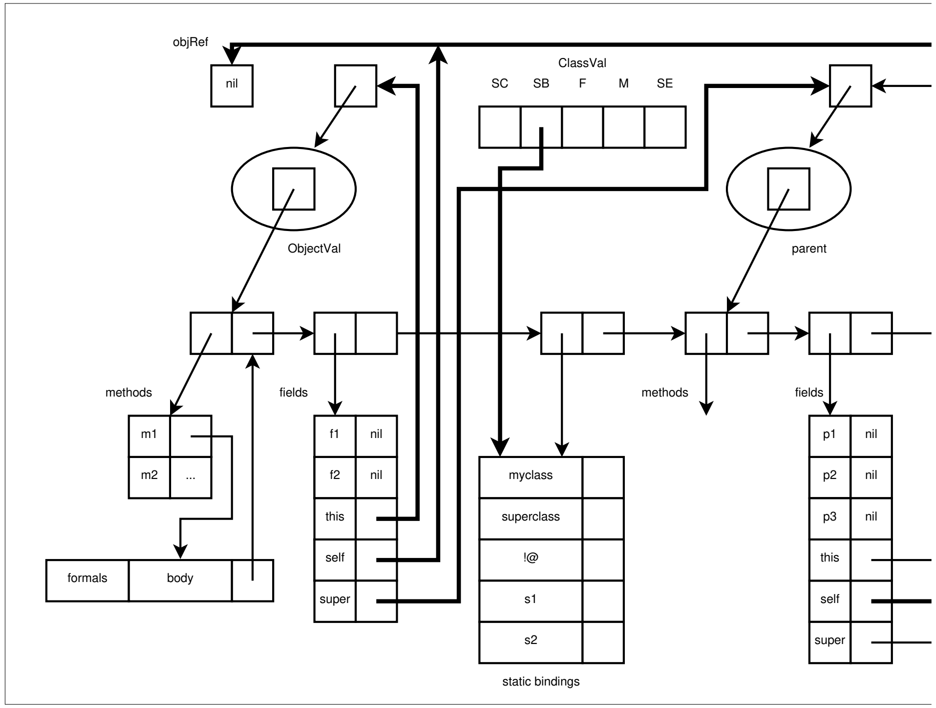
Step 4: Extend the environment of Step 3 with new field bindings



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

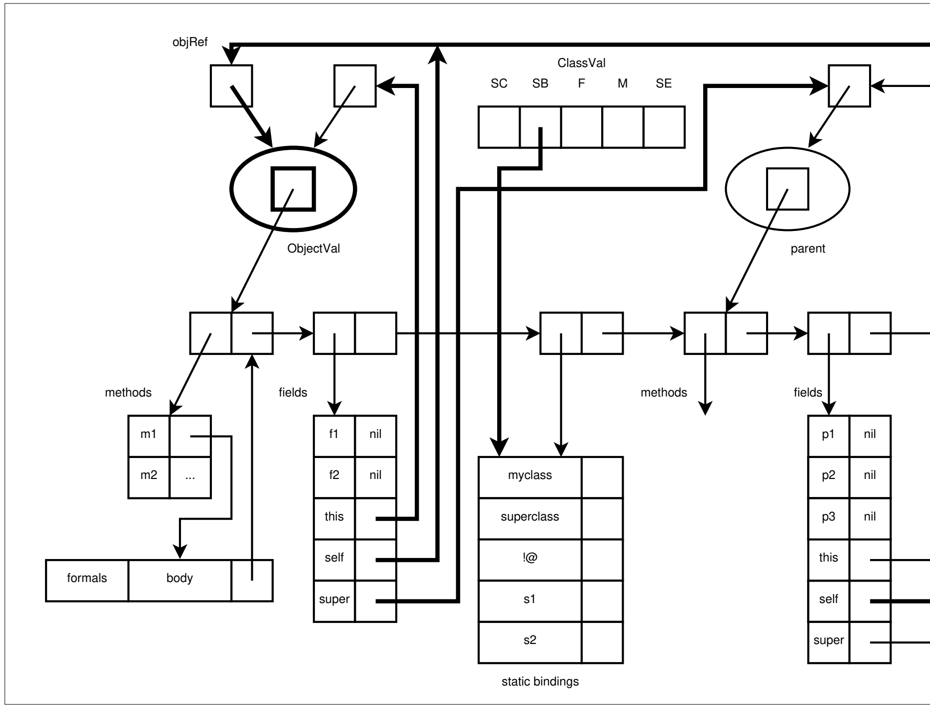
Language OBJ (continued)

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



Language OBJ (continued)

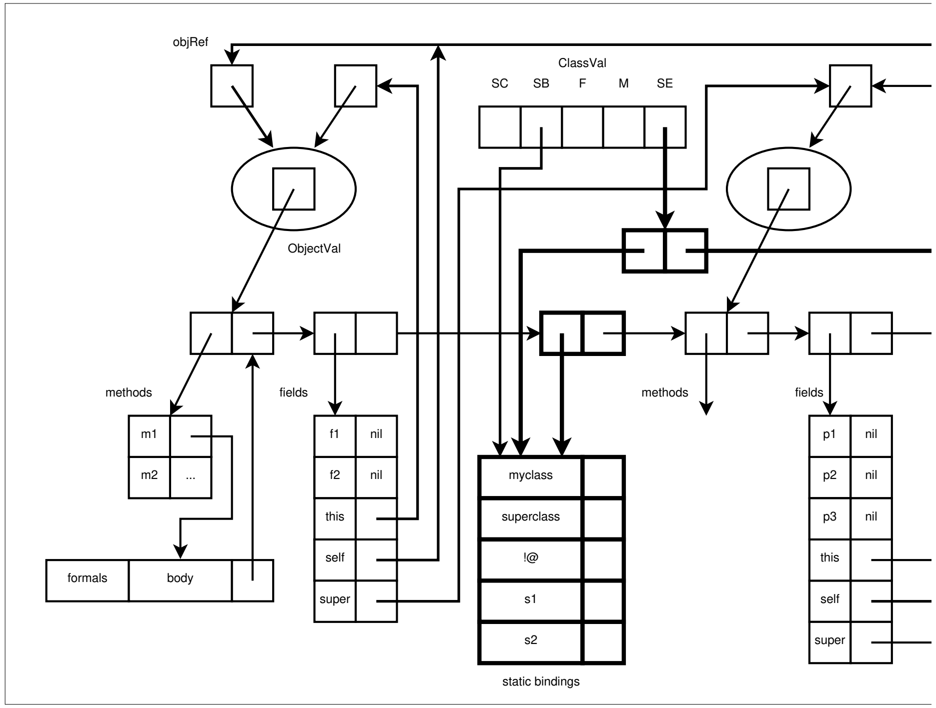
Step 6: Set `objRef` to refer to the newly created object



The `ObjectVal` instance is the value returned by the `new` operato

Language OBJ (continued)

This diagram shows both the object environments and the static env



Observe that the class static bindings belong to both environments.

Language OBJ (continued)

The RHS expressions in static definitions are evaluated in the static e
the class, which extends the static environment of its superclass (an
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 stati
sion can be found in the static environment of the superclass c, wh
to 5. Thus the variable y is bound to 5, not 3, so the expression eval

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 stati
sion can be found in the static environment of superclass c. But c
a variable x, so the value of x must be found in the static environme
class, the singleton EnvClass that captures the top-level environm
the top-level environment is bound to 3, the variable y is bound to 3
expression evaluates to 3.

Language OBJ (continued)

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

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

the expression exp is evaluated in the static environment of the gi
can be used to evaluate an expression that refers to any static variab

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

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

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

Language OBJ (continued)

To evaluate the expression .<c>f(...), we evaluate the procedu
vironment determined by <c>, whereas we evaluate the the actual
pressions in the environment in which the application .<c>f(...
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
given in the static definition of c, but its actual parameter x evaluate
actual parameter expression is evaluated in the local let environm
words, .<c>f(x) is the same as .{<c>f}(x).

Language OBJ (continued)

In particular, observe that the following two expressions are not equ

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

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

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

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

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

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

Language OBJ (continued)

The static environment of a class ultimately ends up extending the
gram environment, not the environment in which the class is define
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
ment extends the top-level environment, not the `let` environment,
this expression is 3, not 5.

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

```
<!@>exp
```

which evaluates to the value of the expression `exp` in the local envi

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 i
`let` environment, and so the environment captured by `!@` has `x` b
above expression can also be written as follows:

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

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

Language OBJ (continued)

The special operator `@` returns an object that captures the *curren*
whatever that may be. (Recall your homework assignment that intr
tation.) This operator may be used to pass a captured environment
to a procedure application or to assign it to a variable for later refer

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

%%%
```

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

Notice that `@` is meaningful in *any* expression context – because e
is evaluated in *some* environment – but that `!@` is only meaningful i
that appears in the context of a class, and its value represents an e
tirely 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 tc
```

Language OBJ (continued)

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

```
<@>exp
exp
```

Language OBJ (continued)

Unlike the `new` operator in Java, Our `new` operator does not take a
and all of the fields are initialized to `nil`. We can, of course, init
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
      end
  in
    let
      o = .<new c>init(3,4)
    in
      <o>+(x,y) % => 7
```

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

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

Language OBJ (continued)

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

```
define c =
    class
        field x
        method init = proc() {set x = 5 ; s
        method foo = proc() {set x = add1(x
    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 follo 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 i in the current environment; this value's `toString()` represe displayed on standard output, and the value is returned as the `DisplayExp` expression. `Display1Exp` is like `DisplayExp` displayed value is followed by a single space. The value of a `New` pression 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 `nil`):

```
3
5
8
```

If the `newline` expressions are removed, the output appears as follows (omittin

```
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 ; se
  end
```

Here's an example of how this class might be used to find and disp
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"
plish the same thing:

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

Language OBJ (continued)

Since we often find ourselves encountering expressions like the foll

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

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 expr
must evaluate to an environment (class or object) – followed by a se
or more procedure applications, each preceded by '>'. Each procedu
in the environment of the previous class or object, and the proced
itself must return another object, whose environment is then used
next procedure, and so on. The entire expression is terminated b
token, defined as '!>'. Note that the actual parameter expressions a
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> RPARE
                  Mangle(List<Exp> expList, List<Rands>
```

The BNF identifier `mangle` should suggest “multiple angle (bracke
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> RPA
                    Mangle(List<Exp> expList, List<Ranc
```

Here is the implementation of how to evaluate an `EenvExp` expres

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

The expression `exp` is evaluated in the current environment. Its valu
the current environment, is passed to the `mangle` object, which the
subsequent procedure applications.

Language OBJ (continued)

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

A `mangle` object consists of a list of `Exp` objects and a list of `Ran`

Language OBJ (continued)

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

```
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 bu
        Val p = expIter.next().eval(v.env());
        // evaluate this method's rands in env
        List<Val> valList = randsIter.next().evalRar
        v = p.apply(valList);
    }
    return v;
}
```

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

Language OBJ (continued)

It turns out that exposing the environment of a procedure can be usec
with procedures alone, a simplified approach to objects and method:

```
OBJ/Prog/pobj
```

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

Language OBJ (continued)

In method application, `self` always refers to the base object, even i
in the definition of a superclass method. This is what makes dy
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
    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 underes
  end

define r = .<new rectangle>init(4,5) % a rectangle with length
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]],[300,35,3
```

Language PROP

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

Suppose, for example, we provided a special designator called served to identify a field whose value was inaccessible outside of the 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 first is that every private field we want to access needs a getter an second is that code such as `<cc>set x = 5` does not work, but is and understand than `.<cc>set_x(5)`.

Language PROP (continued)

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

Here is the same class as described on the previous slide, with a pro|
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 cann
directly except through the property.

The PROP language is based on the OBJ language with the addit
reference semantics and *properties*, as we proceed to describe.

Language PROP (continued)

Here are the grammar rules for defining properties in a class definitio

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

When a variable bound to a property is evaluated, its getExp co
using the environment captured where the property is defined, and
value of the variable is the result of evaluating the getExp express

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

Language PROP (continued)

If a variable `z` [for example] is bound to a property whose `set` exp
side-effects [such as `nil`], an expression such as `set z = ...` d
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-na
evalutes to a thunk-like object called a `PropRef` (which extends t
this object captures the environment in which the property is defin
the property's `get` and `set` expressions.

The expressed value of a variable bound to a property is the valu
evaluating the property's `get` expression in the captured environm
the thunk's `deRef` method.

Similarly, when assigning a value to a variable bound to a proper
`setRef` method is called, with the assigned value bound to the s
'`$`' 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
of this object uses the `PropRef` instead of the variable. Consider t

```
define c =
  class
    static p = proc(t) set t=add1(t)
    field x
    method init = proc(x) {set <this>x=x ;
    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
reference parameter passing semantics, the variable `t` is bound to
and the expression `set t=add1(t)` is evaluated using this bindi
that the RHS parts of property definitions are all evaluated in the en
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,
a role in object instantiation. It turns out that the behavior of pr
be useful even outside of the context of an object, especially to ma
variables defined in a `let` expression. To make this explicit, we crea
construct that has the following concrete syntax and abstract class st

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

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 a
the `set` operator to the variable results in a runtime exception. In th
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
you may recall, is also read-only), so we have the benefit of call-by-n
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
```