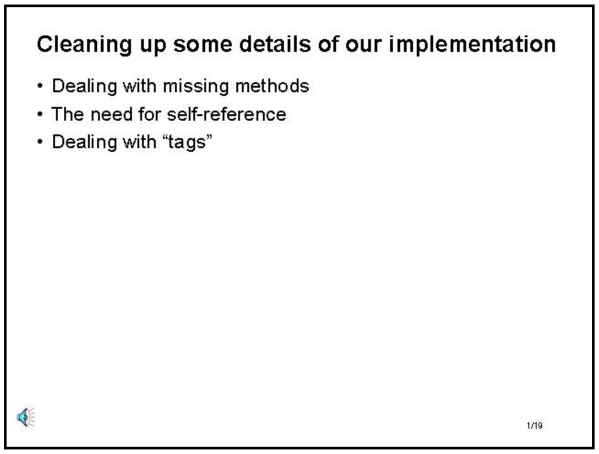
6.001 Notes: Section 14.4

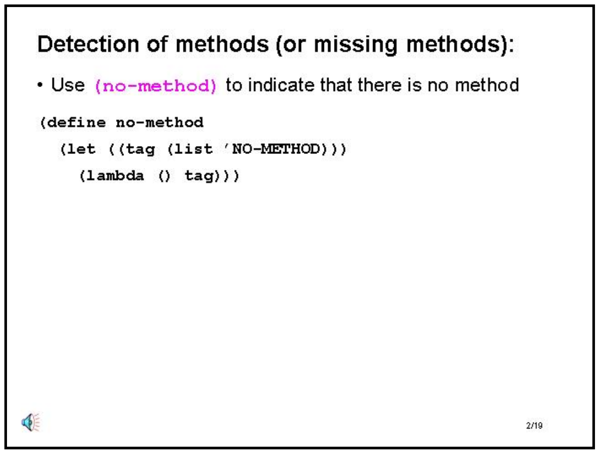
Slide 14.4.1
So we have seen a first pass at building an object-oriented system, using Scheme as the base. There are a few details that we still need to clean up however. These include what to do if a class does not have a method to handle some request, the need to be able to refer to the object within methods belong to an object (i.e. an ability to recursively use methods of an object within an object), and the need to identify types of objects.



Cleaning up some details of our implementation

- Dealing with missing methods
- The need for self-reference
- Dealing with "tags"

1/19



Detection of methods (or missing methods):

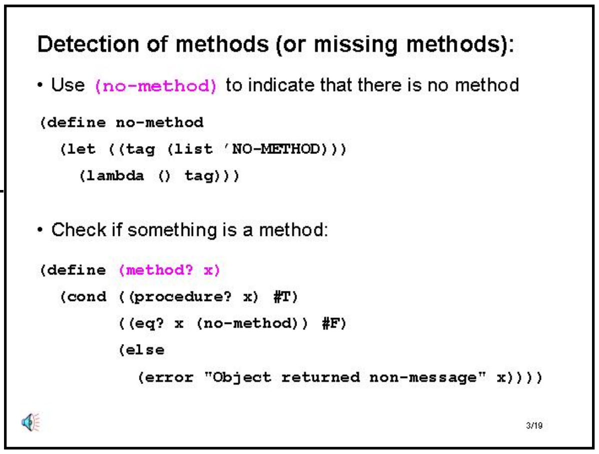
- Use (no-method) to indicate that there is no method

```
(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))
```

2/19

Slide 14.4.2
What happens if the object doesn't know how to handle a message? We need a way to detect if we actually have a method available.
To do this, we first need a kind of tagged system that will tell us when we have no method. But remember our convention: anytime we ask an object for something, it should return a procedure. Thus our way of saying no method exists also needs to be a procedure, as shown with `no-method`, a procedure of no arguments with access to a local frame enclosing a special symbol.

Slide 14.4.3
Then, to check if we have a method, we take an object, such as might be returned by `ask` and we do the following: First, is the argument a procedure? If it is, we assume it is a method, and we proceed. If it is not, we check to see if the value passed in is a `no-method`. We do this by applying `no-method` to get out the tag, then checking to see if the argument is `eq?` to it. If it is, then we return false to indicate that no method exists for this object and message. Otherwise, we signal an error.
So why go through all of this? We need to distinguish between an error in our implementation and an error in trying to get an instance to handle a message it doesn't know about. This lets us separate the details of the implementation from use of the implementation.



Detection of methods (or missing methods):

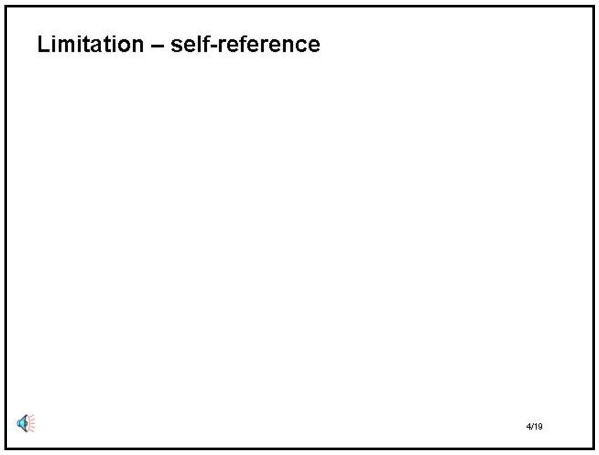
- Use (no-method) to indicate that there is no method

```
(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))
```

- Check if something is a method:

```
(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else
          (error "Object returned non-message" x))))
```

3/19

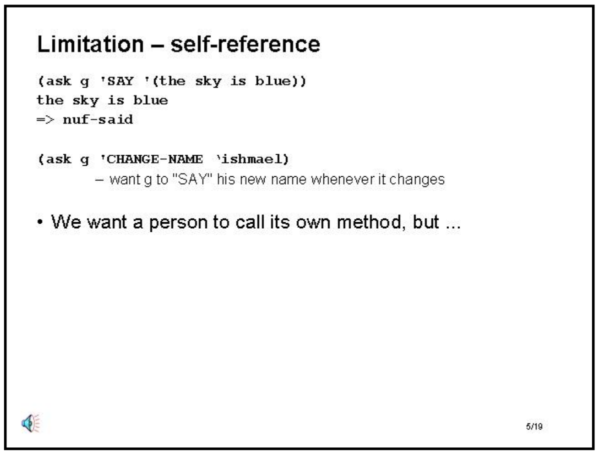


Limitation – self-reference

4/19

Slide 14.4.4
We have now seen an example of creating a class definition, (one of these `maker-` procedures), and we have seen an example of using that class definition to create a particular instance (in this case a person). We have also seen how to separate out getting methods from actually invoking those methods to execute an action. And we have seen how to build a generic interface to the objects, so that we uniformly ask any object to do anything.

Slide 14.4.5
So now we can use this. We can ask `g` to say "The sky is blue", with the behavior shown.
We can also ask the object its name, and we can ask the object to change its name, as shown. But in this latter case, it would be nice if we could get the object to "say" its new name whenever it changed its name. So doing the mutation to change the variable binding is easy. But how do we get a person to use it's own method, that is, how do we get a person to recursively, within one of its methods, ask itself to do something (e.g. SAY)?

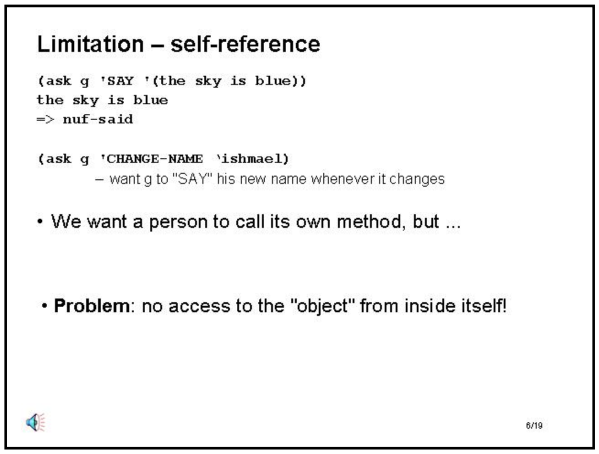


Limitation – self-reference

```
(ask g 'SAY '(the sky is blue))
the sky is blue
=> nuf-said

(ask g 'CHANGE-NAME 'ishmael)
```
– want g to "SAY" his new name whenever it changes

• We want a person to call its own method, but ...

5/19



Limitation – self-reference

```
(ask g 'SAY '(the sky is blue))
the sky is blue
=> nuf-said

(ask g 'CHANGE-NAME 'ishmael)
```
– want g to "SAY" his new name whenever it changes

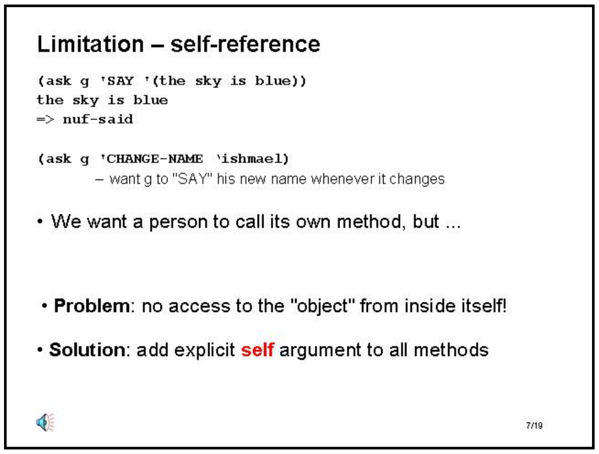
• We want a person to call its own method, but ...

• **Problem**: no access to the "object" from inside itself!

6/19

Slide 14.4.6
Well, we have a problem here. In particular, we don't have inside the code of this object any access to the object itself. We have nothing that points to the procedure representing the object, which would allow us to "ask" it to do something.

Slide 14.4.7
What we need is an **explicit** reference to the object itself. Our way of doing this is to add an explicit argument (called `self`) to all our methods. The goal in doing this is to allow us to have an object be able to refer to itself, so that not only can it do something within a method, it can ask itself to get other methods to do execute other actions.



Limitation – self-reference

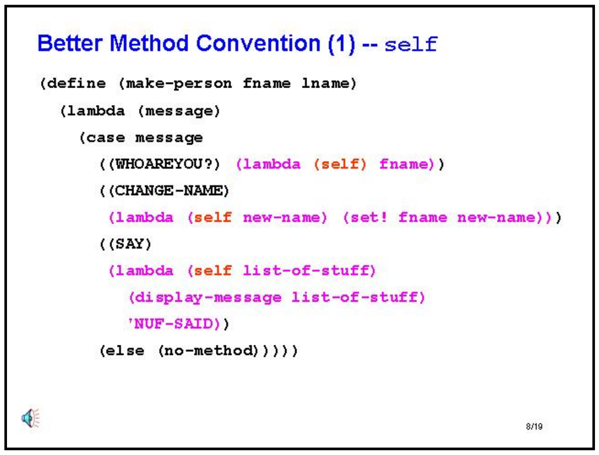
```
(ask g 'SAY '(the sky is blue))
the sky is blue
=> nuf-said

(ask g 'CHANGE-NAME 'ishmael)
```
– want g to "SAY" his new name whenever it changes

• We want a person to call its own method, but ...

• **Problem**: no access to the "object" from inside itself!

• **Solution**: add explicit **self** argument to all methods

7/19



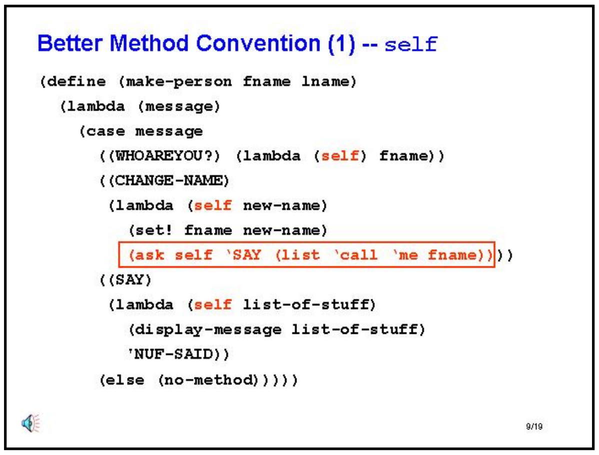
Better Method Convention (1) -- self

```
(define (make-person fname lname)
  (lambda (message)
    (case message
      ((WHOAREYOU?) (lambda (self) fname))
      ((CHANGE-NAME)
       (lambda (self new-name) (set! fname new-name)))
      ((SAY)
       (lambda (self list-of-stuff)
         (display-message list-of-stuff)
         'NUF-SAID))
      (else (no-method)))))
```

8/19

Slide 14.4.8
So here is the first change needed to make this happen. In our class definition, we ensure that each method has a `self` argument as its first argument, by convention.

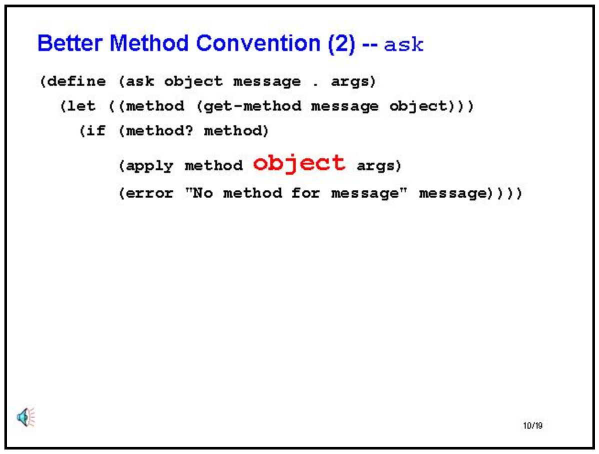
Slide 14.4.9
The second modification is within the `CHANGE-NAME` method. We want the object to ask itself to say its new name. Thus, we `ask` the `self` object to handle a `SAY` message. Notice that this should then ask the same procedure, the procedure representing the object, to handle this new message, which should in turn return a new method for doing exactly that.



Better Method Convention (1) -- self

```
(define (make-person fname lname)
  (lambda (message)
    (case message
      ((WHOAREYOU?) (lambda (self) fname))
      ((CHANGE-NAME)
       (lambda (self new-name)
         (set! fname new-name)
         (ask self 'SAY (list 'call 'me fname))))
      ((SAY)
       (lambda (self list-of-stuff)
         (display-message list-of-stuff)
         'NUF-SAID))
      (else (no-method)))))
```

9/19



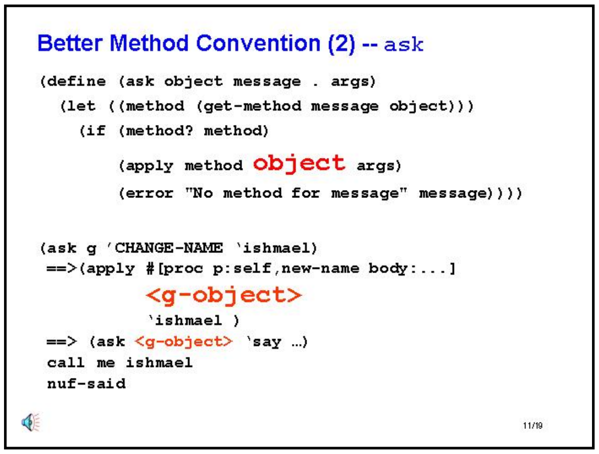
Better Method Convention (2) -- ask

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method object args)
        (error "No method for message" message))))
```

10/19

Slide 14.4.10
Of course we will also need to modify `ask`. This is easy since we just need to explicitly include the object as an argument when applying the method, since each method expects an object in that place in its argument list. Notice an important design issue here. By separating out `ask` from other parts of the system, and especially separating out the idea of getting a method from the idea of applying it, we have made it easier to incorporate changes such as this one.

Slide 14.4.11
So how does this change give us more capabilities in our system, especially our ability to ask an object to do something within the context of another method? Suppose we evaluate `(ask g 'CHANGE-NAME 'ishmael)`. This will first get the method for changing names from `g`. That returned procedure is then applied to the object that corresponds to `g` (i.e. the value bound to `g` in this environment) and the argument `ishmael`. This will first mutate the binding for `fname` in this environment, from `george` to `ishmael`. And then it will ask this object; the value associated with `g` or more particularly

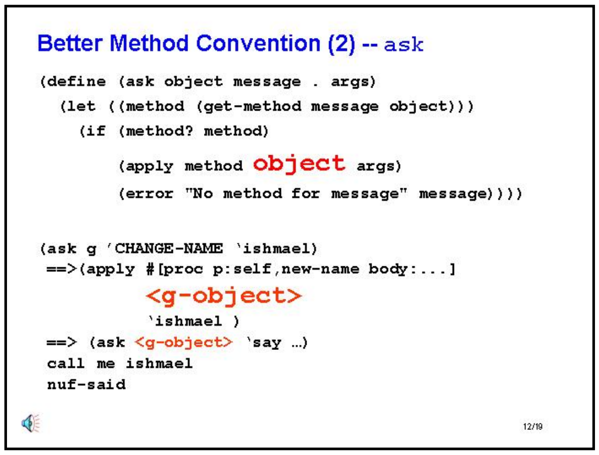


Better Method Convention (2) -- ask

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method object args)
        (error "No method for message" message))))

(ask g 'CHANGE-NAME 'ishmael)
 ==>(apply #[proc p:self,new-name body:...]
           <g-object>
           'ishmael )
 ==> (ask <g-object> 'say …)
 call me ishmael
 nuf-said
```

11/19

the value associated with `object` in this frame, to "say" something.



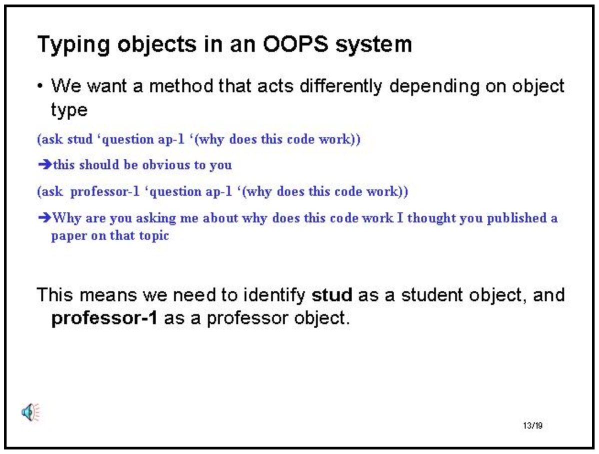
Better Method Convention (2) -- ask

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method object args)
        (error "No method for message" message))))

(ask g 'CHANGE-NAME 'ishmael)
 ==>(apply #[proc p:self,new-name body:...]
           <g-object>
           'ishmael )
 ==> (ask <g-object> 'say …)
 call me ishmael
 nuf-said
```

12/19

Slide 14.4.12
As before, let's step back from the details to consider what is being accomplished here. We have been designing an interface for objects. We have seen that we can use a generic `ask` procedure to get methods for instances, but we have also seen that in some cases we want the methods to interact with one another. This requires letting an object be able to refer to itself within a method. This led to the extension to our design that we just went through. While this was a small change in terms of code, it was a big change in terms of impact on behavior of the system.

Slide 14.4.13
One other detail that we need to consider is how to identify the type of an object. Remember in our earlier example, we wanted to add a method to an **arrogant-prof** so that he would respond in one manner if the person asking a question was a **student**, and in a different manner if the person asking a question was a **professor**. How do we add the equivalent of "type tags" to our objects in an object-oriented system?



Typing objects in an OOPS system

- We want a method that acts differently depending on object type

(ask stud 'question ap-1 '(why does this code work))

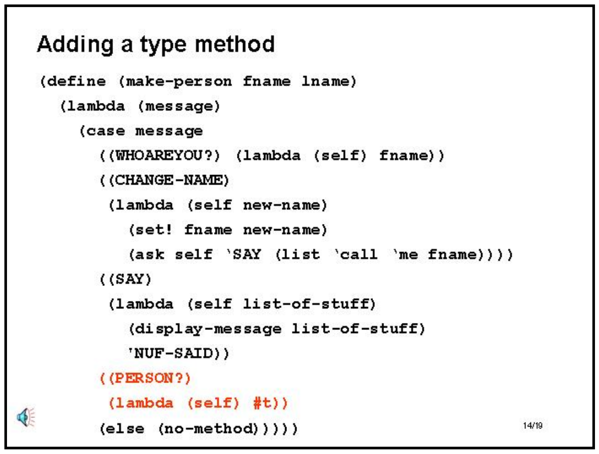
➔this should be obvious to you

(ask professor-1 'question ap-1 '(why does this code work))

➔Why are you asking me about why does this code work I thought you published a paper on that topic

This means we need to identify **stud** as a student object, and **professor-1** as a professor object.

13/19



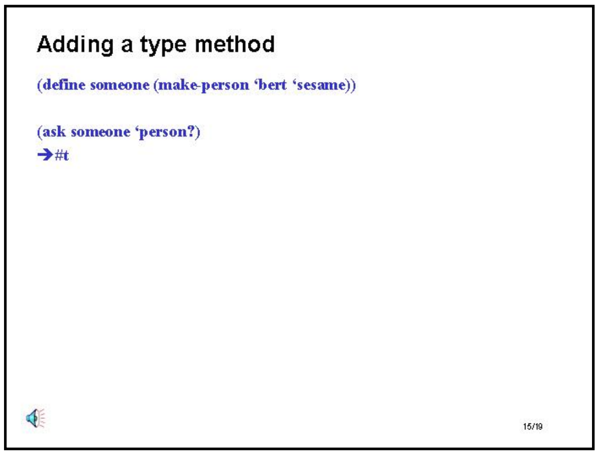
Adding a type method

```
(define (make-person fname lname)
  (lambda (message)
    (case message
      ((WHOAREYOU?) (lambda (self) fname))
      ((CHANGE-NAME)
       (lambda (self new-name)
         (set! fname new-name)
         (ask self 'SAY (list 'call 'me fname))))
      ((SAY)
       (lambda (self list-of-stuff)
         (display-message list-of-stuff)
         'NUF-SAID))
      ((PERSON?)
       (lambda (self) #t))
      (else (no-method)))))
```

14/19

Slide 14.4.14
One easy way to do this is to use the basic component of an object, namely a method. Here is an example for our **person** class definition.

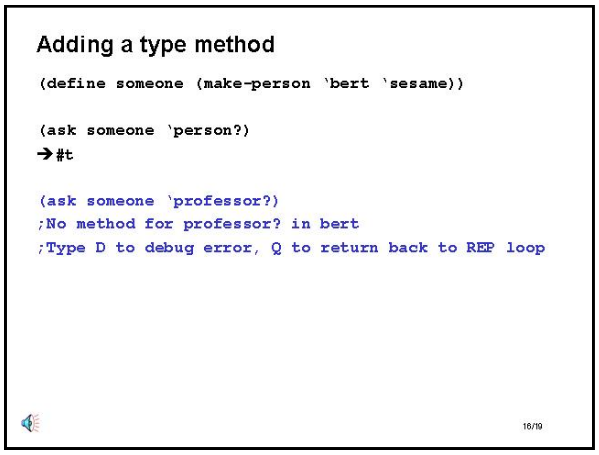
Slide 14.4.15
Now, if we check to see if someone is a person, we get the response we expect. Clearly we could write methods for classes in which an argument is checked for its type, and different behaviors are used based on that type.



Adding a type method

```
(define someone (make-person 'bert 'sesame))

(ask someone 'person?)
➔#t
```

15/19



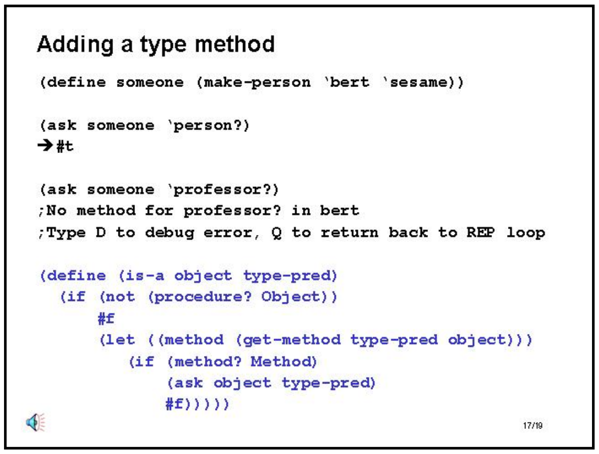
Adding a type method

```
(define someone (make-person 'bert 'sesame))

(ask someone 'person?)
➔#t

(ask someone 'professor?)
;No method for professor? in bert
;Type D to debug error, Q to return back to REP loop
```

16/19

Slide 14.4.16
But we need to be careful! If we ask an object about a different type, we get an error (due to a lack of method) rather than the behavior we wanted, namely returning false to say that the object is not of this type.

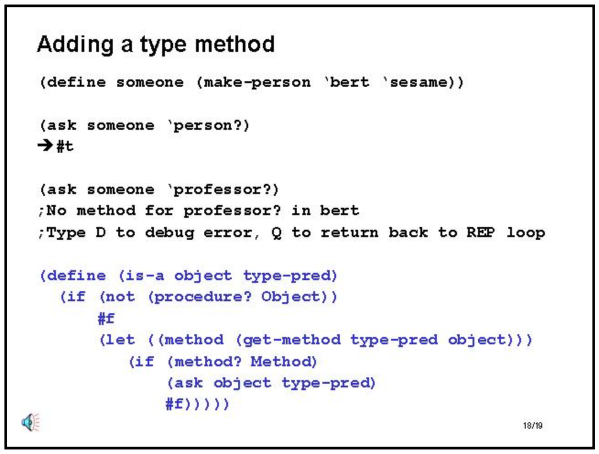
Slide 14.4.17
To fix this, we need to add one more detail, namely a way of asking if an object is of a particular type. This is shown here. This procedure can be used to check in an object is of a particular type, but will return true or false, rather than failing due to a lack of method.



Adding a type method

```
(define someone (make-person 'bert 'sesame))

(ask someone 'person?)
➔#t

(ask someone 'professor?)
;No method for professor? in bert
;Type D to debug error, Q to return back to REP loop

(define (is-a object type-pred)
  (if (not (procedure? Object))
      #f
      (let ((method (get-method type-pred object)))
         (if (method? Method)
             (ask object type-pred)
             #f)))))
```

17/19



Adding a type method

```
(define someone (make-person 'bert 'sesame))

(ask someone 'person?)
➔#t

(ask someone 'professor?)
;No method for professor? in bert
;Type D to debug error, Q to return back to REP loop

(define (is-a object type-pred)
  (if (not (procedure? Object))
      #f
      (let ((method (get-method type-pred object)))
         (if (method? Method)
             (ask object type-pred)
             #f)))))
```

18/19

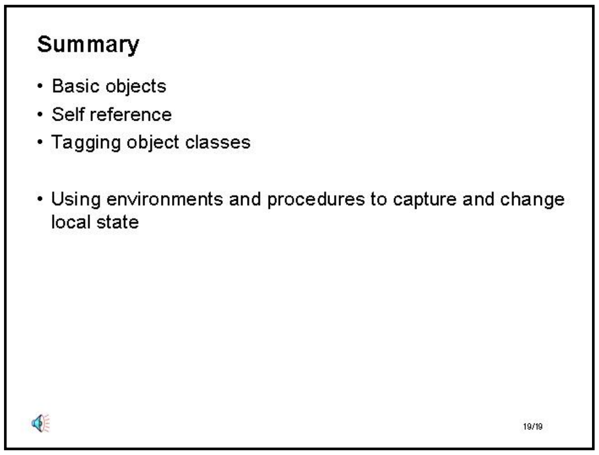
Slide 14.4.18
The point of this last example is to show that we can add the same abilities we saw earlier with tagged data types. We simply need to ensure that our mechanism for checking type tags is consistent with the object-oriented framework. But with this ability, we can now inherit all the power we saw earlier of dispatching on type, in this case to different objects.

Slide 14.4.19

Thus, we have seen how to create (in Scheme) a system for describing simple object-oriented frameworks. The system includes a means of defining classes, a means of creating instances of those classes, and ways of referring to instances of classes, including oneself.

We have seen how we can use the idea of an environment to capture local state, and to use procedures created on demand to provide methods that can access and change the state of instances of classes.

In the next lecture, we will turn to more complex ways of creating and using classes, especially the issues of inheritance and delegation.



Summary

- Basic objects
- Self reference
- Tagging object classes

- Using environments and procedures to capture and change local state

19/19

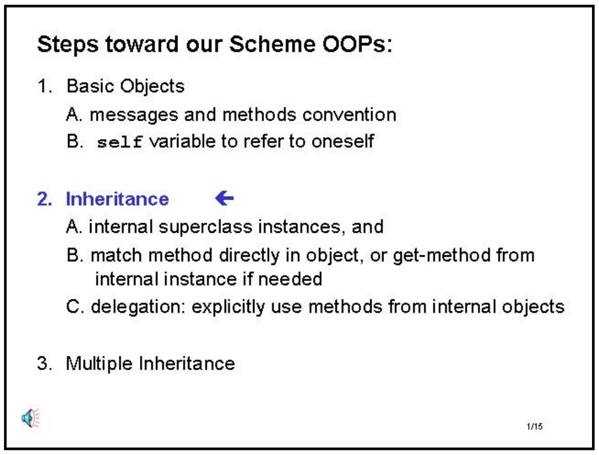
6.001 Notes: Section 14.5

Slide 14.5.1

In the last part of this set of lectures, we looked at the basic elements of object-oriented programming. We examined the role of objects as a paradigm for structuring systems, and we saw how we could use our knowledge of Scheme to construct a framework for building classes and instances in an object-oriented system.

So where are we? We have established ways of creating classes and instances in our object oriented system, as well as conventions for dealing with messages and methods, and conventions for allowing objects to refer to themselves to support methods calling other methods.

Now we want to look at using these ideas to explore the idea of inheritance. Recall that **inheritance** meant having the ability to create subclasses of objects, or specialized classes of objects, which could inherit behaviors from the superclass of objects. The goal was to have different specializations of a general class so that the common methods between specializations could be captured in the superclass, and the variations of unique methods could be isolated within subclasses.



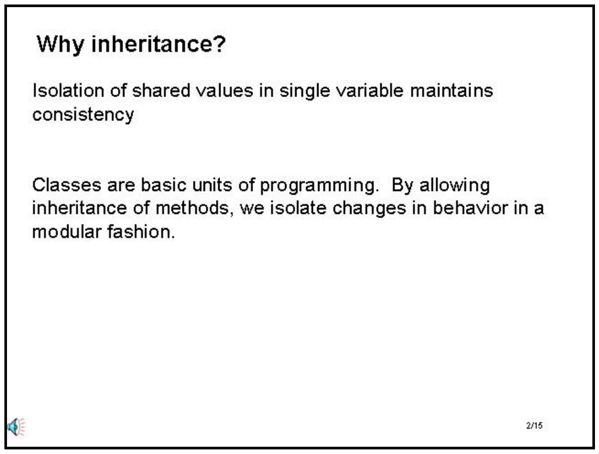
Steps toward our Scheme OOPs:

1. Basic Objects
   A. messages and methods convention
   B. `self` variable to refer to oneself

2. Inheritance ←
   A. internal superclass instances, and
   B. match method directly in object, or get-method from internal instance if needed
   C. delegation: explicitly use methods from internal objects

3. Multiple Inheritance

1/15



Why inheritance?

Isolation of shared values in single variable maintains consistency

Classes are basic units of programming. By allowing inheritance of methods, we isolate changes in behavior in a modular fashion.

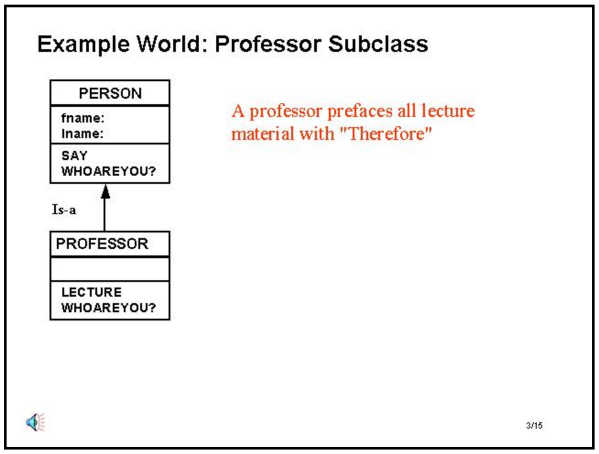
2/15

Slide 14.5.2

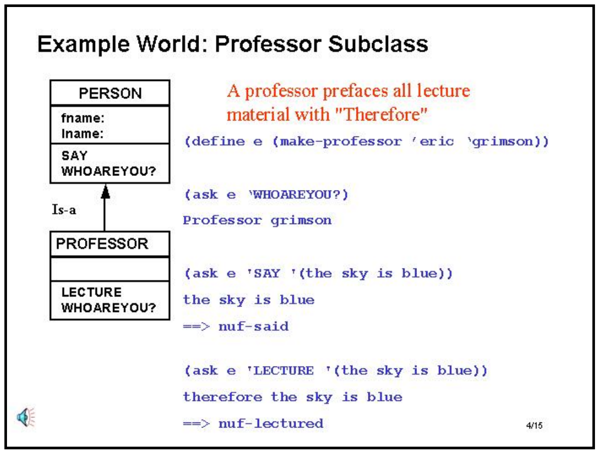
Now why would we want the ability to inherit, within a hierarchy of classes? First, by isolating a shared value in a single variable, we make it easier to maintain consistency of values. In other words, we avoid having multiple versions of the same variable, and thus isolate changes in that variable to a single location.

Second, under this new view of programming, classes become our basic units of code construction. As a consequence, we would like to enforce modularity as much as possible on classes, and by enabling the construction of hierarchies of classes, in which a subclass can inherit methods (as well as variables) from superclass instances, we ensure consistency of behavior and isolate changes to a single method.

Slide 14.5.3
Let's extend our current example to look at this issue. We already have a class, called a **person.** We can create a subclass of person, called a **professor**. A professor, because it is a kind of person, has the same capabilities as a person. It has an internal variable for its names; it has methods for returning its name, for changing its name, and for saying things. However, a professor has a unique capability, different from normal people. When a professor is "lecturing", it prefaces all said material with the word "Therefore". Thus it has a new method, `LECTURE`, which does that.


Example World: Professor Subclass

A professor prefaces all lecture material with "Therefore"


Example World: Professor Subclass

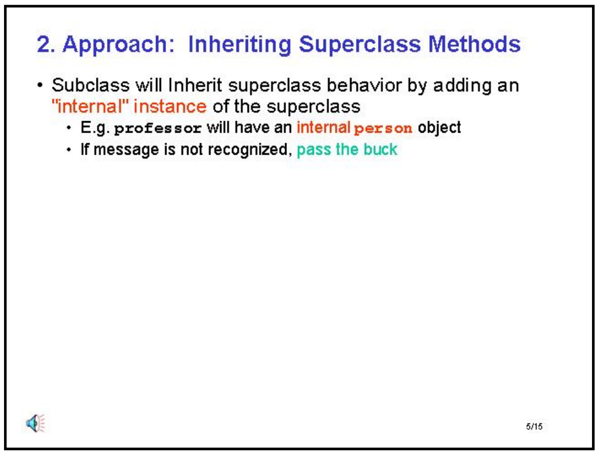
A professor prefaces all lecture material with "Therefore"

```
(define e (make-professor 'eric 'grimson))

(ask e 'WHOAREYOU?)
Professor grimson

(ask e 'SAY '(the sky is blue))
the sky is blue
==> nuf-said

(ask e 'LECTURE '(the sky is blue))
therefore the sky is blue
==> nuf-lectured
```

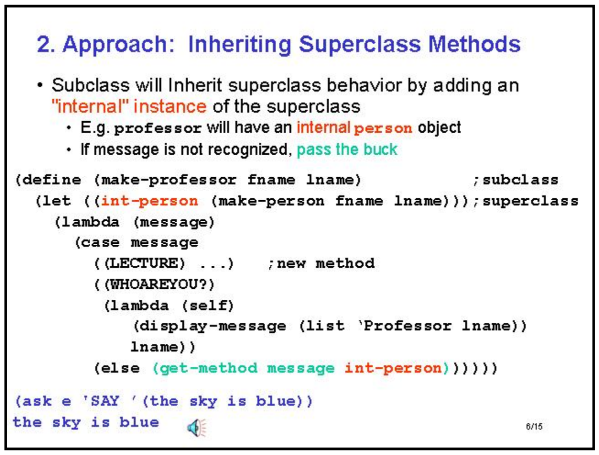
PERSON
fname:
lname:
SAY
WHOAREYOU?
Is-a
PROFESSOR
LECTURE
WHOAREYOU?
4/15

Slide 14.5.4
"Therefore" the behavior we expect is the following: If we define `e` in the global environment to be an instance of a professor, using a `make-` procedure that we will discuss shortly, then we can ask `e` to `SAY` things. In this case, it behaves just like a person, as it inherits the `SAY` method from its superclass.
If we ask `e` its name, we get a different behavior from a normal person, which suggests that we will need a new method that shadows the person’s method.
Finally, we can ask `e` to `LECTURE` about the chromaticity of the atmosphere. In that case, `e` as a professor should use the `LECTURE` method. This will, as a consequence, say `therefore the sky is blue`.
So we see that this object `e` should both have the ability to use methods specific to a professor and the ability to inherit methods from an underlying person.

Slide 14.5.5
Here is the approach we will take to make this happen. In particular, we are going to allow the inheritance of methods from superclasses to subclasses, by within each subclass creating an internal instance of a superclass. In other words, a professor will have within it an internal instance of a person. Then, if a message is passed to the professor, the professor will first see if it has an explicit method for that message. If so, it does its thing. If not, it "passes the buck" to the internal person instance, asking it to handle this message.


2. Approach: Inheriting Superclass Methods

- Subclass will Inherit superclass behavior by adding an "internal" instance of the superclass
  - E.g. `professor` will have an internal `person` object
  - If message is not recognized, pass the buck



2. Approach: Inheriting Superclass Methods

- Subclass will Inherit superclass behavior by adding an "internal" instance of the superclass
  - E.g. `professor` will have an internal `person` object
  - If message is not recognized, pass the buck

```
(define (make-professor fname lname)           ;subclass
  (let ((int-person (make-person fname lname)));superclass
    (lambda (message)
      (case message
        ((LECTURE) ...)    ;new method
        ((WHOAREYOU?)
         (lambda (self)
            (display-message (list 'Professor lname))
            lname))
        (else (get-method message int-person))))))

(ask e 'SAY '(the sky is blue))
the sky is blue
```

6/15

Slide 14.5.6

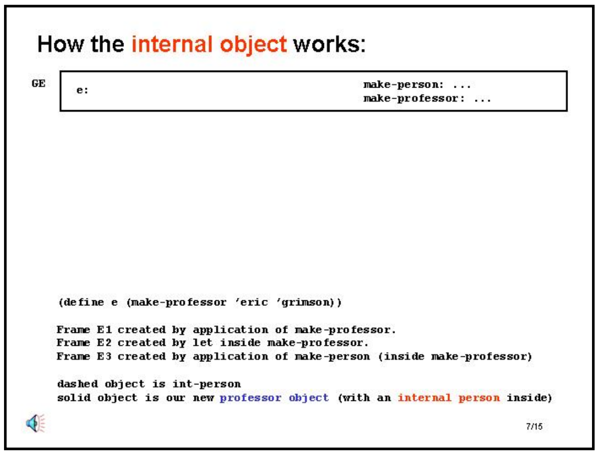
Using that idea, we can then build a `make-` procedure for professors, that is, we can implement the class definition of professors. Recall that it is going to be a subclass, which is going to inherit from persons. Note how the constructor works. It creates an internal person, i.e., it literally calls the `make-` procedure for persons and creates a binding for `int-person` to that internal instance. It then returns an object that represents an instance of this class. As before, it is a message-passing procedure. Here, it has one thing it explicitly knows how to do: to `LECTURE`. Otherwise, it gets the method for the message from its internal instance of a person. This means it literally passes "the buck" back to the internal person, saying "You figure out how to handle this message and return a method for me".

Notice the form used. This object satisfies our convention, as it will return a method (or procedure) for all messages. But the default, rather than saying there is no method, is to ask the superclass instance to handle things. Thus, if we ask `e` to `SAY` something, since `e` is a professor it will first look for an explicit `SAY` method, and then deducing it doesn't have one, it will ask its internal person to `SAY`. This will, as we saw in the earlier slides, then return a method for saying things, and apply it.

Slide 14.5.7

So let's trace this through. We will suppress some of the details of the environment model, so that we can see the general flow of computation.

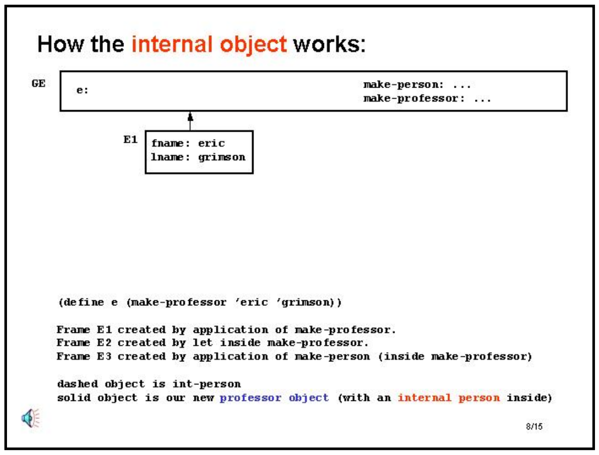
Defining `e` to be a professor using the appropriate `make-` procedure should have the following behavior.



How the internal object works:

```
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

7/15



How the internal object works:

```
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

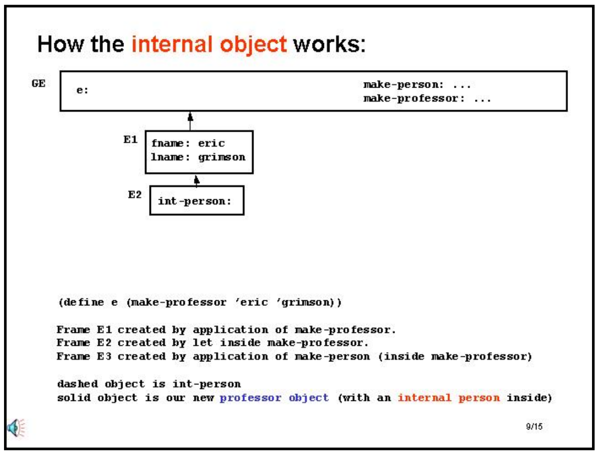
8/15

Slide 14.5.8

First, applying `make-professor` will drop a frame in which the variable `fname` is bound to the symbol `eric` and the variable `lname` bound to the symbol `Grimson.` Thus, frame E1 is created by the application of `make-professor`.

Slide 14.5.9

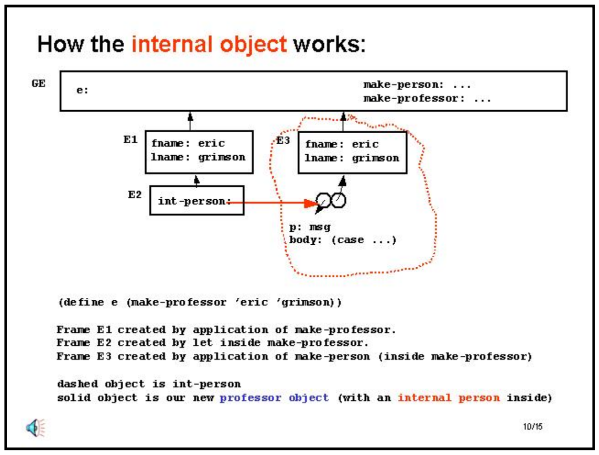
Recall that the body of `make-professor` has within it a `let` expression. You have seen that evaluating a `let` will cause a new frame to be created that is scoped by the current frame. Within that frame, we bind `int-person` to some value. Thus, frame E2 is created by the evaluation of the `let` within `make-professor`, and is scoped by E1.



How the internal object works:

```
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

Slide 14.5.10



How the internal object works:

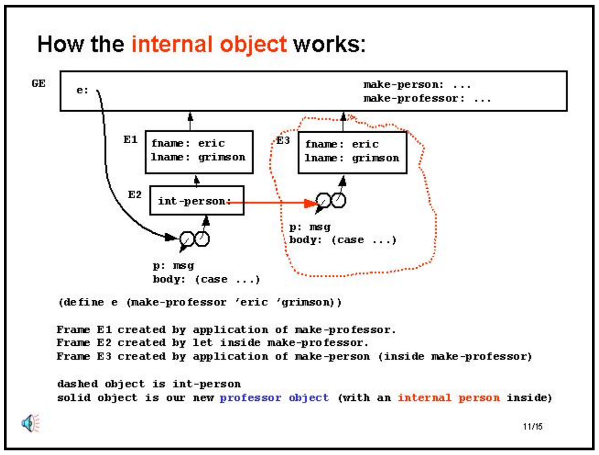
```
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

... and what is `int-person` bound to? … to the result of evaluating `(make-person ...)`. And we know what that does as we saw it earlier. It creates a new frame through the application of `make-person` and notice that this frame is scoped by the global environment, because that is where the `make-person` procedure's environment pointer points to. Within that frame we bind the variable `fname` to the argument passed in, and relative to that frame we evaluate the body of `make-person` which returns a message-passing object corresponding to an instance of a person. `Int-person` is then bound to this object, as that is the value returned by the application of `make-person`.

Slide 14.5.11

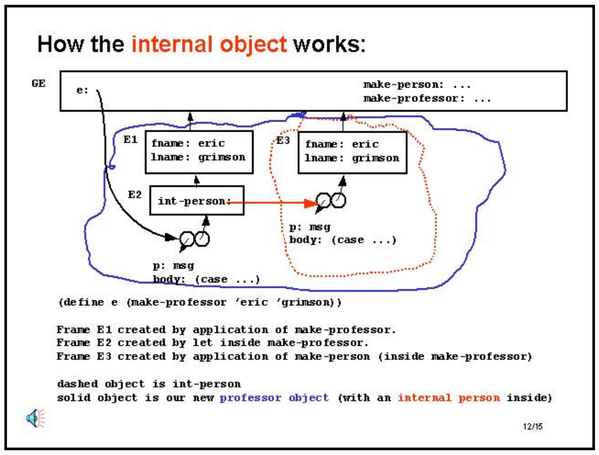
Finally, we evaluate the body of the `let` expression inside `make-professor`, with respect to E2 (remember that was the frame created by evaluating the first part of the let). That creates the message-passing object corresponding to an instance of a person. It's environment pointer points to this frame, and the procedure is the value returned by the application of `make-professor`. Therefore, `e` is bound to this value in the global environment.

This has a structure that is very useful in our system!



How the internal object works:

```
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

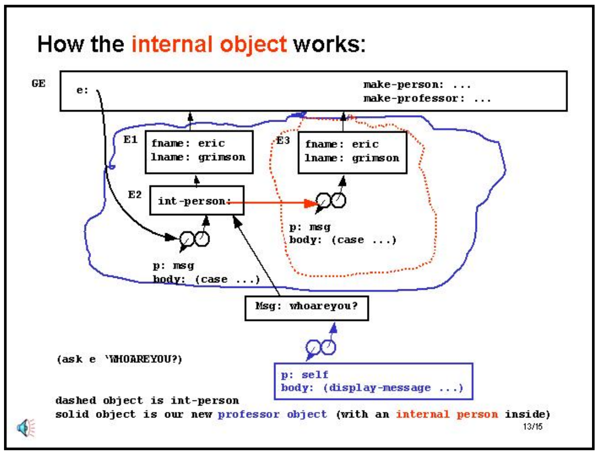

How the internal object works:

```
(define e (make-professor 'eric 'grimson))

Frame E1 created by application of make-professor.
Frame E2 created by let inside make-professor.
Frame E3 created by application of make-person (inside make-professor)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

12/15

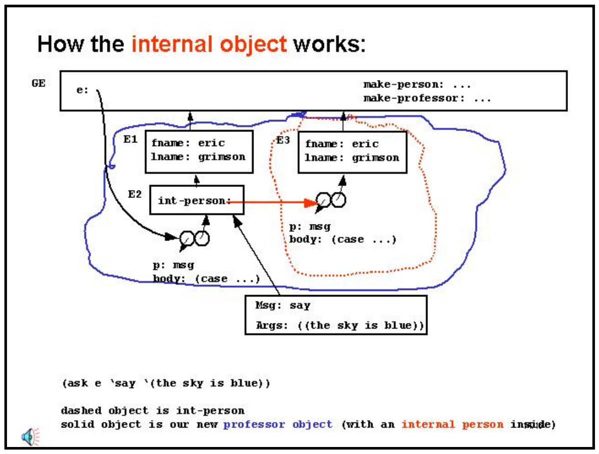
Slide 14.5.12
In particular, `e` in the global environment points to a structure: a professor object (the thing enclosed in blue), which is a procedure with access to local frames. Within that object, there is an object, an internal person (the thing enclosed in red). Notice that it is referred to by a local variable that points to one of these structures: a message-passing object that has access to some local frames. Thus, we have an internal instance within another instance, and this will support the idea of inheritance.
All we have to do is specify how the top-level object will pass along requests for methods to the internal object.

Slide 14.5.13
First, suppose we ask this object to identify itself. In this case, the process of "ask"ing will eventually cause the system to apply the object `e` to the message `whoareyou?` Because the **professor** class has an explicit method for handling this message, it will create a procedure relative to this frame. And since this frame is scoped by the frames created during the construction of the object, the procedure will have access to the internal variables, and can thus identify the object for us.


How the internal object works:

```
(ask e 'WHOAREYOU?)

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

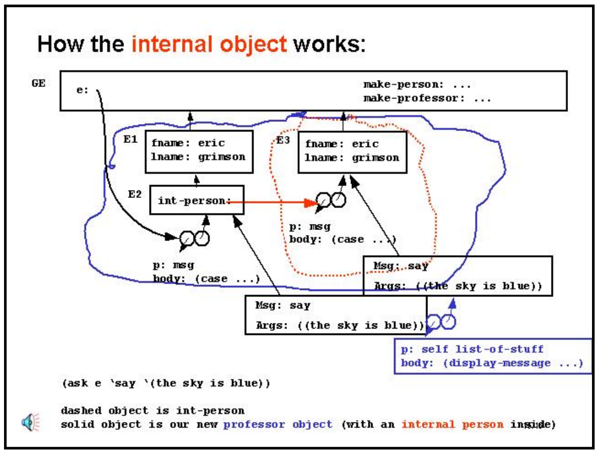
13/15


How the internal object works:

```
(ask e 'say '(the sky is blue))

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

Slide 14.5.14
But suppose we ask this object to say something. In this case, the process of "ask"ing will again cause the system to apply the object `e` to the message `say.` Because the **professor** class does **not** have an explicit method for handling this message, it will try to inherit a method from its internal **person** object.

Slide 14.5.15
This means that the same message will be sent to the object that corresponds to the internal instance of the superclass. As a consequence, we will create a method for handling this request with respect to this frame, and then proceed.
Thus we see that an object can inherit methods from internal instances of superclasses. As a consequence, if we decide to change how a class handles a method, we need only do it in the definition of that class, and the changed behavior will then be automatically inherited by subclasses.

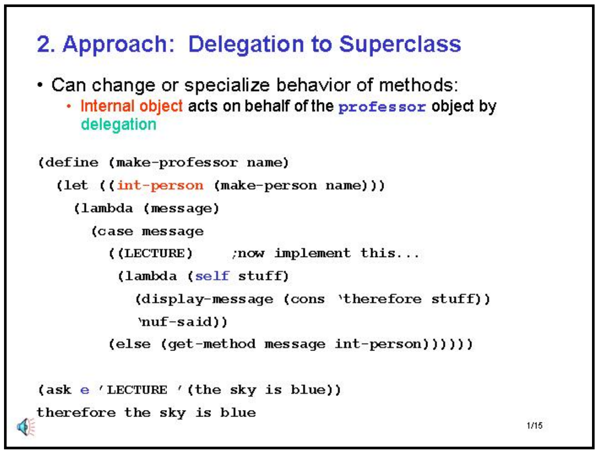

How the internal object works:

```
(ask e 'say '(the sky is blue))

dashed object is int-person
solid object is our new professor object (with an internal person inside)
```

6.001 Notes: Section 14.6

Slide 14.6.1

So we have seen how we can create subclasses: procedures that accept messages as before, have access to internal state, but include within them a pointer to another object that belongs **only** to that instance, but that has information or capabilities of the superclass. Now let's look at how that provides power in controlling behaviors of objects.
First, suppose we want our professor to **lecture.** We said that "lecturing" simply meant adding the word "therefore" to the beginning of each utterance, so here is a simple way of accomplishing this.

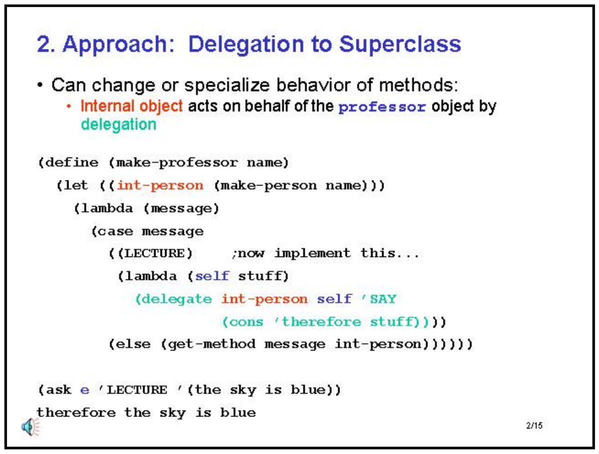


2. Approach: Delegation to Superclass

- Can change or specialize behavior of methods:
  - Internal object acts on behalf of the `professor` object by delegation

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (lambda (message)
      (case message
        ((LECTURE)    ;now implement this...
         (lambda (self stuff)
           (display-message (cons 'therefore stuff))
           'nuf-said))
        (else (get-method message int-person))))))

(ask e 'LECTURE '(the sky is blue))
therefore the sky is blue
```

1/15



2. Approach: Delegation to Superclass

- Can change or specialize behavior of methods:
  - Internal object acts on behalf of the `professor` object by delegation

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (lambda (message)
      (case message
        ((LECTURE)    ;now implement this...
         (lambda (self stuff)
           (delegate int-person self 'SAY
                     (cons 'therefore stuff))))
        (else (get-method message int-person))))))

(ask e 'LECTURE '(the sky is blue))
therefore the sky is blue
```

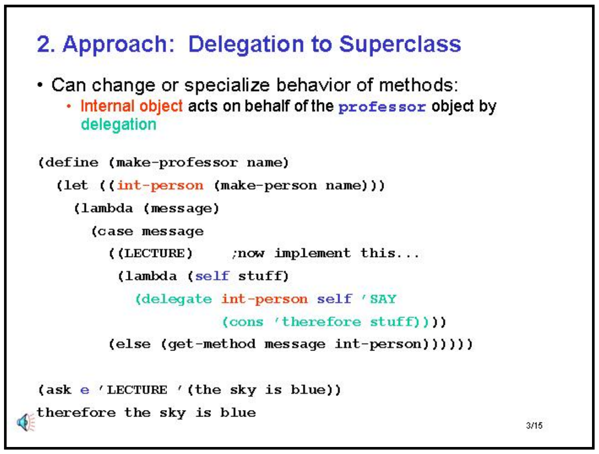
2/15

Slide 14.6.2

But, a little thought suggests that at the level of code and implementation, there is a significant overlap in the code to implement **lecture** in a **professor** and the code for **say** in **person.** As well, it seems reasonable as a statement about our "world" that **lecture**ing is actually a form of **say**ing. That is, the overlap is not just an accident of code, but rather these two methods concern actions that are variants of one another at the conceptual level.
So we want to indicate that a **professor lecture**ing is a variant on a **person say**ing something, both on the implementation level and on the conceptual level. Object-oriented programming offers a way to acknowledge both of these cases: it's called ***delegation***, and the idea is that a **professor lecture**ing is done by having it delegate (or hand off) the job to its superclass (**person**) and requesting it to **say** the right thing.
Here is the change we make to capture this behavior

Slide 14.6.3

When we want our professor to `lecture` we really just want him/her to `say` the word "therefore" followed by whatever else he/she was going to say. So our method for `LECTURE` is a procedure (or method) with an argument, `self`, in order to be able to refer to the object, plus the set of things to be said. This procedure will then ask the internal person to `SAY` the word "therefore" followed by whatever else he/she was going to say. Thus, we would like to **delegate** to the internal person a request to say the appropriate stuff. And if we ask a professor to lecture "the sky is blue" the internal person would be asked to say "therefore the sky is blue".



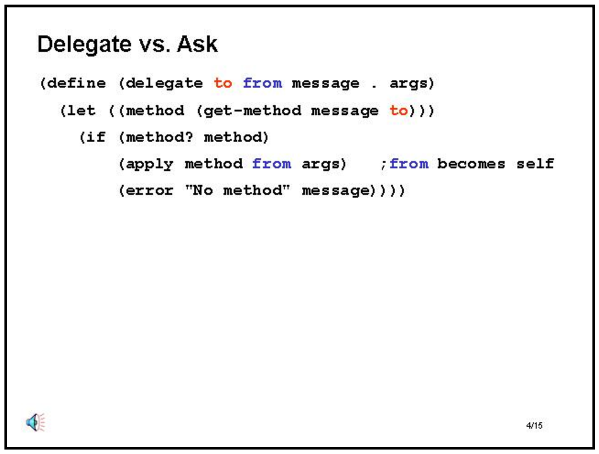
2. Approach: Delegation to Superclass

- Can change or specialize behavior of methods:
  - Internal object acts on behalf of the `professor` object by delegation

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (lambda (message)
      (case message
        ((LECTURE)    ;now implement this...
         (lambda (self stuff)
           (delegate int-person self 'SAY
                     (cons 'therefore stuff))))
        (else (get-method message int-person))))))

(ask e 'LECTURE '(the sky is blue))
therefore the sky is blue
```

3/15

We will have to implement `delegate` but the idea makes intuitive sense. Delegation would allow us to designate from one object a request to a specific other object to do something. This should result in the other object providing the method needed by the first object to accomplish the desired task.



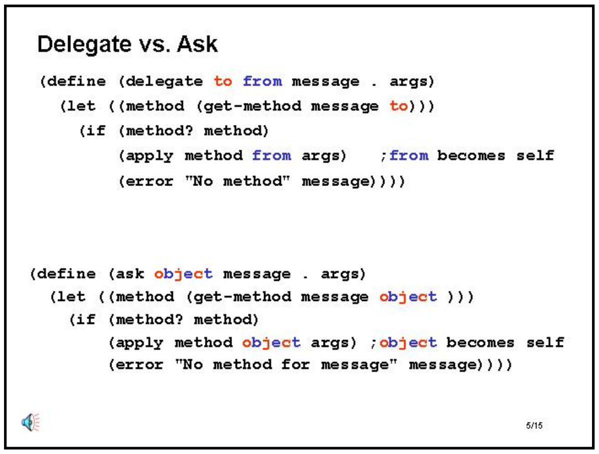
Delegate vs. Ask

```
(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
        (apply method from args)    ;from becomes self
        (error "No method" message))))
```

4/15

Slide 14.6.4
To implement the idea of `delegation` we want to pass a message from one object to another. Notice that we can just get the method of the `to` object associated with the message, then apply that method with the `from` object (the object that asked for this delegation) as the `self` object.

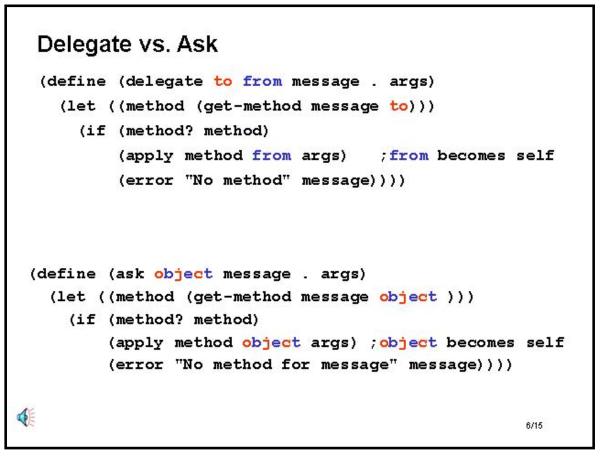
Slide 14.6.5
This looks a lot like `ask`, right? `Ask` took a single object and a message, got the method for that message from that object, and then applied that method to the same object.
`Delegate` extends this to differentiate the object providing the method from the object to which that method is being applied.



Delegate vs. Ask

```
(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
        (apply method from args)    ;from becomes self
        (error "No method" message))))


(define (ask object message . args)
  (let ((method (get-method message object )))
    (if (method? method)
        (apply method object args) ;object becomes self
        (error "No method for message" message))))
```

5/15



Delegate vs. Ask

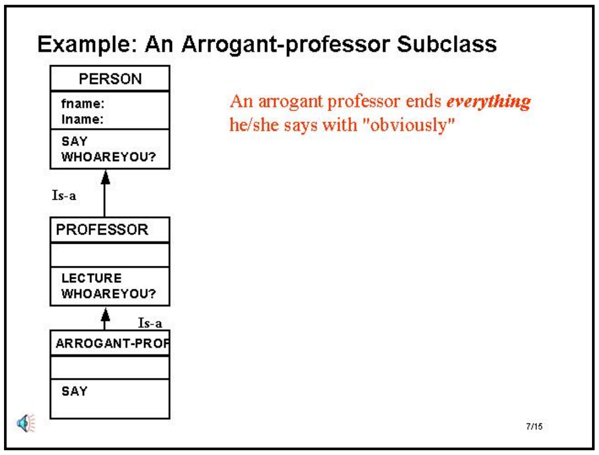
```
(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
        (apply method from args)    ;from becomes self
        (error "No method" message))))


(define (ask object message . args)
  (let ((method (get-method message object )))
    (if (method? method)
        (apply method object args) ;object becomes self
        (error "No method for message" message))))
```

6/15

Slide 14.6.6
What we have done is extend the power of our object-oriented system. We can create subclasses, we can inherit methods from superclasses, and we can delegate specific requests to instances of subclasses. This should greatly increase the range of behaviors we can now simulate.

Slide 14.6.7

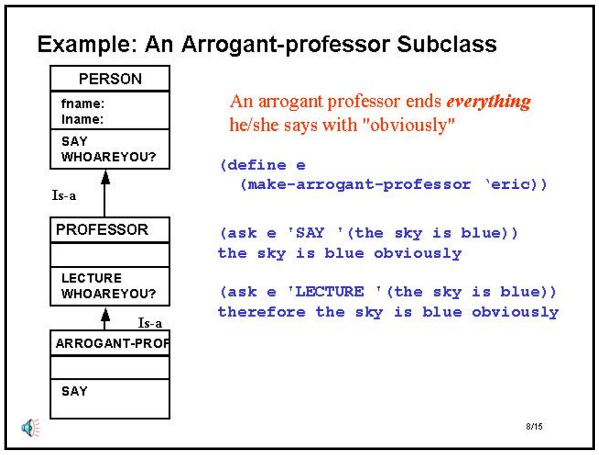
Let's return to our example system and further extend it. We have a person, we have a professor, now let's add a new kind of person, an `arrogant-professor`. This is someone who ends every statement with the word "obviously". We would like this to be a subclass that inherits from a professor which itself inherits from a person, but has a different kind of behavior. We want the arrogant professor, whenever he/she says anything to end it with this word "obviously".



Example: An Arrogant-professor Subclass

An arrogant professor ends *everything* he/she says with "obviously"

7/15



Example: An Arrogant-professor Subclass

An arrogant professor ends *everything* he/she says with "obviously"

```
(define e
  (make-arrogant-professor 'eric))

(ask e 'SAY '(the sky is blue))
the sky is blue obviously

(ask e 'LECTURE '(the sky is blue))
therefore the sky is blue obviously
```

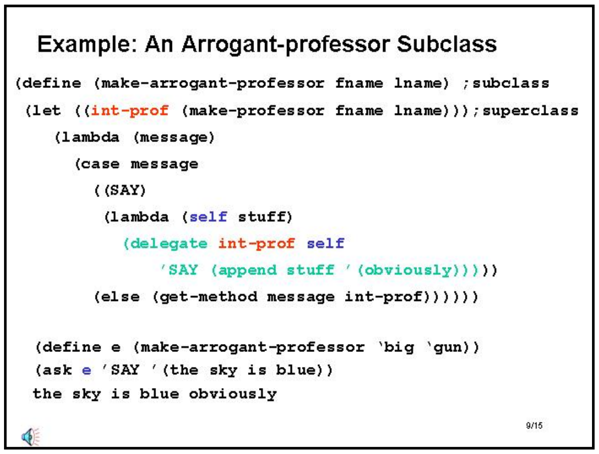
8/15

Slide 14.6.8

If we define `e` in this case to be an instance of an arrogant professor, and ask it to `SAY` "the sky is blue", he then says "the sky is blue obviously" as expected. And if we ask him to `LECTURE` on the topic of the chromaticity of the atmosphere, he says "therefore the sky is blue obviously".

Slide 14.6.9

With this class design in mind, it seems easy to implement this idea using superclasses. Our `make-arrogant-professor` procedure should simply create an internal instance of a professor, which will itself have within it an internal instance of a person. Then the message-passing procedure that represents instances of arrogant professors will simply delegate to the professor, upon receiving a request to `SAY`, the requirement to use the internal `SAY` method of the professor to say the things, with "obviously" added at the end. Of course, the professor should then add a "therefore" to the front of this set of things to say.

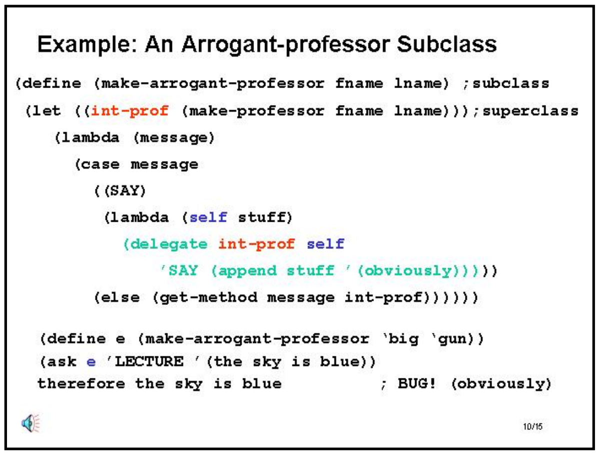


Example: An Arrogant-professor Subclass

```
(define (make-arrogant-professor fname lname) ;subclass
 (let ((int-prof (make-professor fname lname)));superclass
   (lambda (message)
     (case message
       ((SAY)
        (lambda (self stuff)
          (delegate int-prof self
              'SAY (append stuff '(obviously)))))
       (else (get-method message int-prof))))))

 (define e (make-arrogant-professor 'big 'gun))
 (ask e 'SAY '(the sky is blue))
 the sky is blue obviously
```

9/15

Thus this object should be able to say things and to inherit the ability to lecture from the internal instance as well. Let's check it out. Clearly it can say things as we expected.



Example: An Arrogant-professor Subclass

```
(define (make-arrogant-professor fname lname) ;subclass
 (let ((int-prof (make-professor fname lname)));superclass
    (lambda (message)
      (case message
        ((SAY)
         (lambda (self stuff)
           (delegate int-prof self
               'SAY (append stuff '(obviously)))))
        (else (get-method message int-prof))))))

  (define e (make-arrogant-professor 'big 'gun))
  (ask e 'LECTURE '(the sky is blue))
  therefore the sky is blue           ; BUG! (obviously)
```

10/15

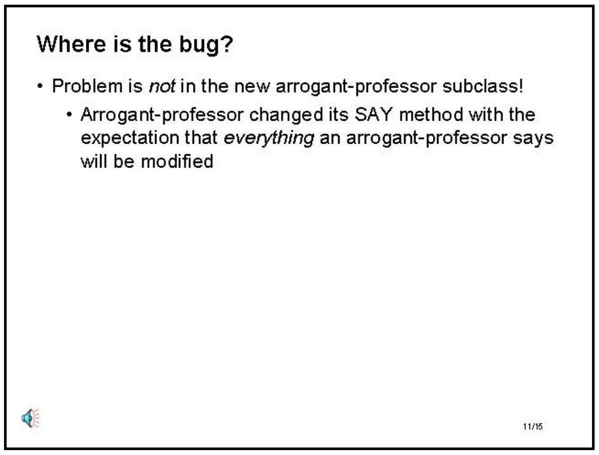
Slide 14.6.10

So I ask `e` to `LECTURE` " the sky is blue" and it says "therefore the sky is blue".

OOPS! Where is the "obviously"? This didn't work! Why?

Slide 14.6.11

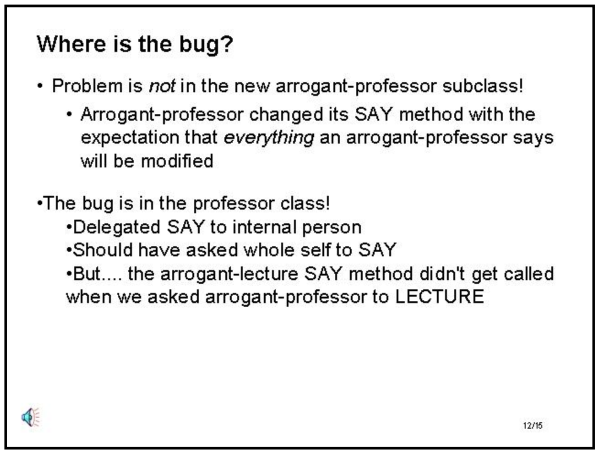
The problem is **not** with the thing we just built! The new arrogant professor subclass did the right thing. Arrogant professor changed its SAY method with the expectation that everything the arrogant professor says will be modified. That's the behavior we want.



Where is the bug?

- Problem is *not* in the new arrogant-professor subclass!
  - Arrogant-professor changed its SAY method with the expectation that *everything* an arrogant-professor says will be modified

11/15



Where is the bug?

- Problem is *not* in the new arrogant-professor subclass!
  - Arrogant-professor changed its SAY method with the expectation that *everything* an arrogant-professor says will be modified

•The bug is in the professor class!
- •Delegated SAY to internal person
- •Should have asked whole self to SAY
- •But.... the arrogant-lecture SAY method didn't get called when we asked arrogant-professor to LECTURE

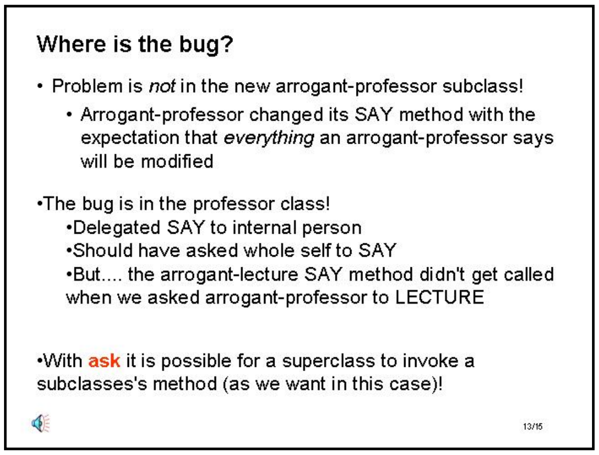
12/15

Slide 14.6.12

But think about what happens. When we ask an arrogant professor to `LECTURE` something, it delegates to its internal professor a request to `LECTURE` ".... obviously". But **that** will then use the internal `SAY` method of the internal `person` and we really should have asked the arrogant professor self to `SAY` this. What is the `SAY` method for the internal person? It just says the passed in argument.

In particular, the `SAY` method associated with the arrogant professor did NOT get called when we asked it to lecture because it delegated this job through to the internal person. Thus it correctly `SAY`s but it incorrectly `LECTURE`s. The problem is that we were not careful in our design of the professor class to say when we wanted to have something delegated.

Slide 14.6.13
The way to fix this behavior is to use `ask` because `ask` will make it possible for a superclass to invoke a subclass' method as we want in this case. Thus, we have two different kinds of behavior mechanisms: delegation and asking.

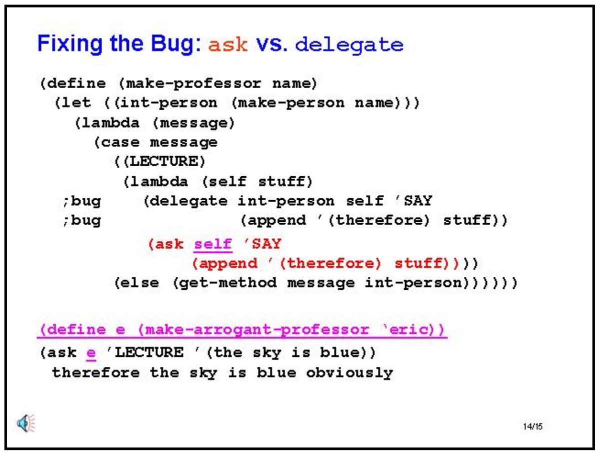

Where is the bug?

- Problem is *not* in the new arrogant-professor subclass!
  - Arrogant-professor changed its SAY method with the expectation that *everything* an arrogant-professor says will be modified

•The bug is in the professor class!
  •Delegated SAY to internal person
  •Should have asked whole self to SAY
  •But.... the arrogant-lecture SAY method didn't get called when we asked arrogant-professor to LECTURE

•With ask it is possible for a superclass to invoke a subclasses's method (as we want in this case)!

13/15

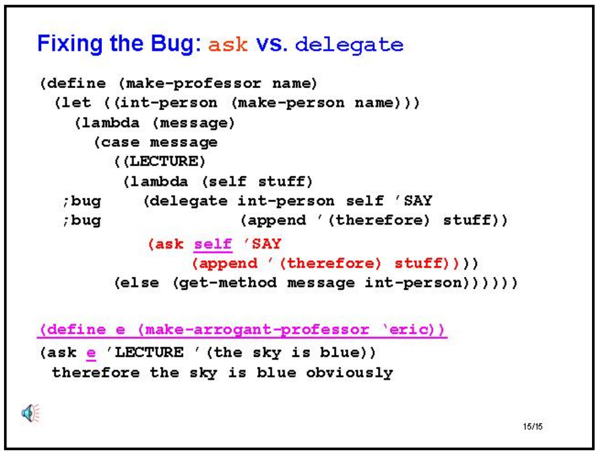

Fixing the Bug: ask vs. delegate

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (lambda (message)
      (case message
        ((LECTURE)
         (lambda (self stuff)
;bug        (delegate int-person self 'SAY
;bug                   (append '(therefore) stuff))
              (ask self 'SAY
                   (append '(therefore) stuff))))
        (else (get-method message int-person))))))

(define e (make-arrogant-professor 'eric))
(ask e 'LECTURE '(the sky is blue))
 therefore the sky is blue obviously
```

14/15

Slide 14.6.14
So in this case, we can accomplish what we want with a simple change. Inside of our `make-professor` procedure we change the behavior. We still have an internal person, but rather than delegating when asked to LECTURE to the internal person, we will ASK the self to SAY with "therefore" at the front.
Thus when we make an arrogant professor, asking it to lecture will use the SAY method of the arrogant lecturer, not the SAY method of the internal person.

Slide 14.6.15
This is a rather subtle point, and the reason we are raising it is to let you see the variations in behavior one can get. One of the interesting challenges in designing object oriented systems is in breaking up the system into the right sized modules and associated behaviors and at the same time controlling the interactions of behaviors between the classes, especially in the presence of inheritance and hierarchies of classes.
As we can see, if there are conflicting or competing methods within these hierarchies, we have to think carefully about asking which object to execute which method.

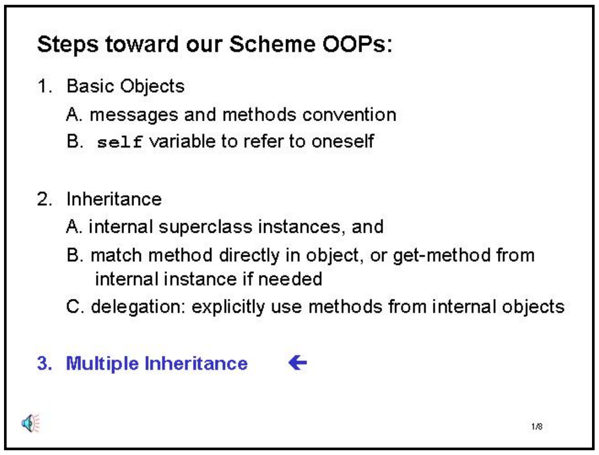

Fixing the Bug: ask vs. delegate

```
(define (make-professor name)
  (let ((int-person (make-person name)))
    (lambda (message)
      (case message
        ((LECTURE)
         (lambda (self stuff)
;bug        (delegate int-person self 'SAY
;bug                   (append '(therefore) stuff))
              (ask self 'SAY
                   (append '(therefore) stuff))))
        (else (get-method message int-person))))))

(define e (make-arrogant-professor 'eric))
(ask e 'LECTURE '(the sky is blue))
 therefore the sky is blue obviously
```

15/15

6.001 Notes: Section 14.7

Slide 14.7.1
Now that we have seen inheritance, the ability for an internal instance of a superclass to provide methods to specializations of objects, what happens when we have multiple inheritance (i.e. what happens when we have objects that inherit methods from different kinds of superclasses)?

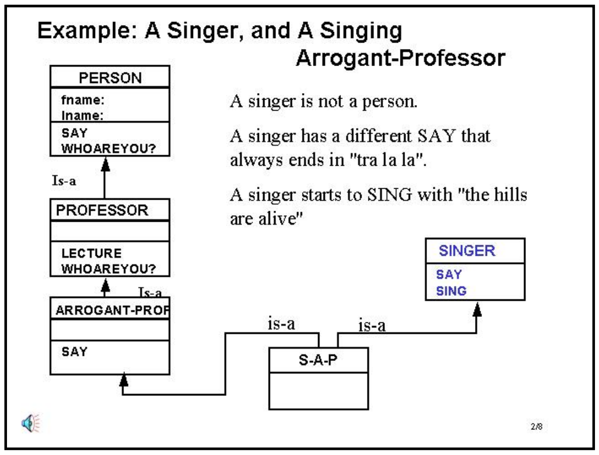

Steps toward our Scheme OOPs:

1. Basic Objects
   A. messages and methods convention
   B. `self` variable to refer to oneself

2. Inheritance
   A. internal superclass instances, and
   B. match method directly in object, or get-method from internal instance if needed
   C. delegation: explicitly use methods from internal objects

3. **Multiple Inheritance** ←

1/8


Example: A Singer, and A Singing Arrogant-Professor

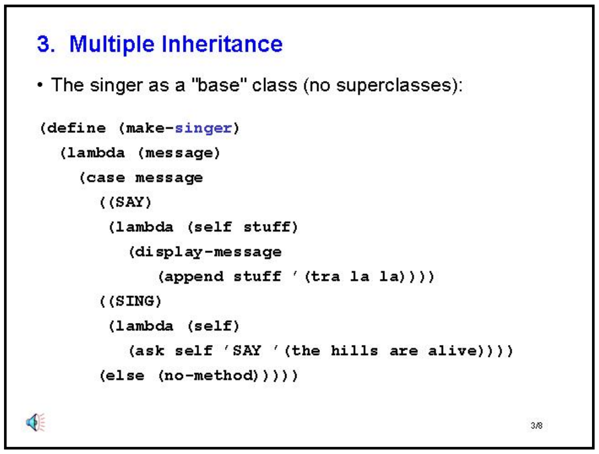
A singer is not a person.

A singer has a different SAY that always ends in "tra la la".

A singer starts to SING with "the hills are alive"

Slide 14.7.2
Let's add a new object, a new class, to our system. A **singer** is distinct from a person. It has its own `SAY` method (which always ends with "tra la la"), as well as having a `SING` method (which starts with "the hills are alive").
On top of this, we can then create a "singing arrogant professor". God knows what it actually does although maybe you have seen of few of these folks around MIT. The idea is that an `s-a-p` should inherit methods from both an arrogant professor and from a singer. This will lead to some interesting questions about how one decides where to inherit a method from, when there are multiple choices of methods.

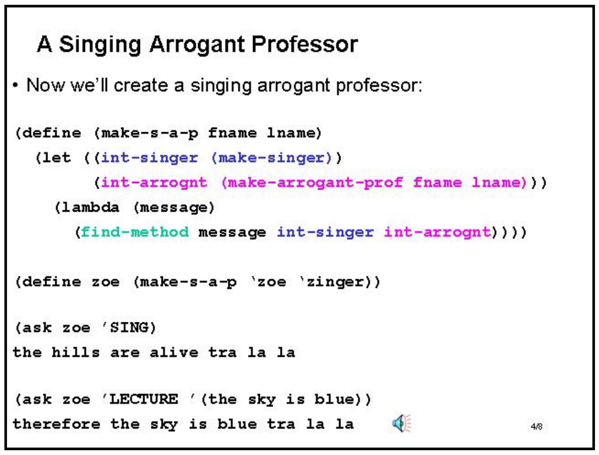
Slide 14.7.3
First, we can build our base representation or base class. There is no superclass here, because a singer is a basic class. The definition for a singer is very simple: it's a message-passing object that handles methods for saying and singing, as shown (noting that singing uses the objects `SAY` method). This is just like our other class definitions in form.


3. Multiple Inheritance

• The singer as a "base" class (no superclasses):

```
(define (make-singer)
  (lambda (message)
    (case message
      ((SAY)
       (lambda (self stuff)
         (display-message
            (append stuff '(tra la la)))))
      ((SING)
       (lambda (self)
         (ask self 'SAY '(the hills are alive))))
      (else (no-method)))))
```

3/8



A Singing Arrogant Professor

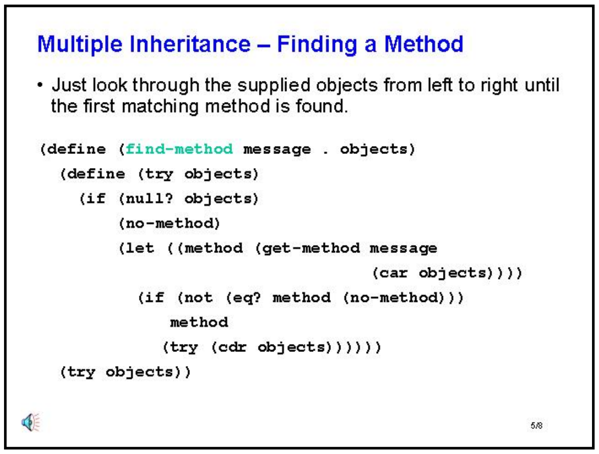
- Now we'll create a singing arrogant professor:

```
(define (make-s-a-p fname lname)
  (let ((int-singer (make-singer))
        (int-arrognt (make-arrogant-prof fname lname)))
    (lambda (message)
      (find-method message int-singer int-arrognt))))

(define zoe (make-s-a-p 'zoe 'zinger))

(ask zoe 'SING)
the hills are alive tra la la

(ask zoe 'LECTURE '(the sky is blue))
therefore the sky is blue tra la la
```

4/8

Slide 14.7.4
Now we can create the class of a singing arrogant professor. We will have within the constructor for this class, something that creates an internal singer, using `make-singer`, something that creates an internal arrogant professor, using `make-arrogant-professor` and we will have references to both of those. Then the object that accepts messages for the singing arrogant professor should simply take the message and find a method, for example by first looking in the singer, then in the arrogant professor if the singer does not have a method. The behavior would then be what we expect as shown in the examples of singing and lecturing.

Slide 14.7.5
To find a method, we will simply look through the objects in order until one returns a method. Thus, `find-method` takes a message and a list of objects, and scans through a loop until it either runs out of objects or gets a returned method.

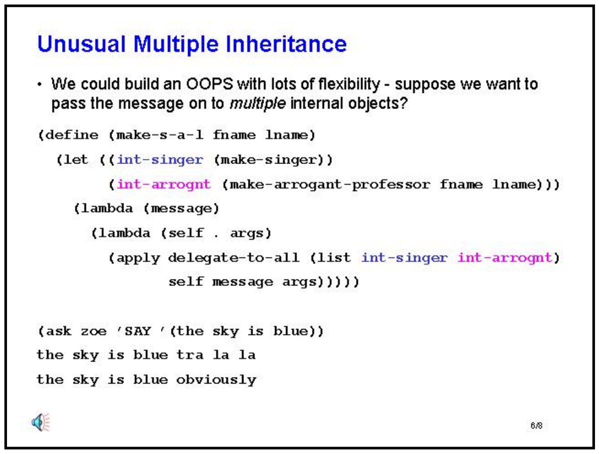


Multiple Inheritance – Finding a Method

- Just look through the supplied objects from left to right until the first matching method is found.

```
(define (find-method message . objects)
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method (get-method message
                                  (car objects))))
          (if (not (eq? method (no-method)))
              method
             (try (cdr objects))))))
  (try objects))
```

5/8



Unusual Multiple Inheritance

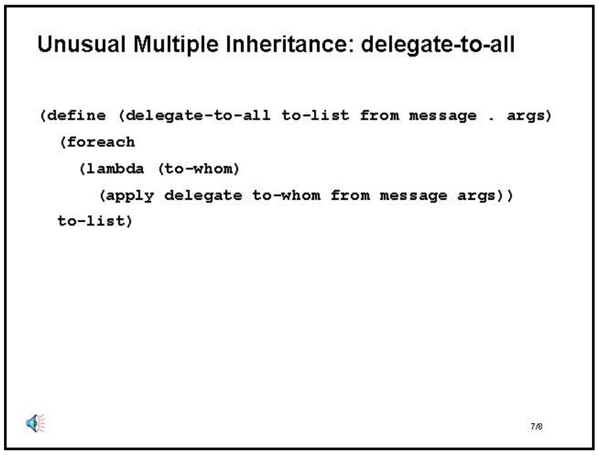
- We could build an OOPS with lots of flexibility - suppose we want to pass the message on to *multiple* internal objects?

```
(define (make-s-a-l fname lname)
  (let ((int-singer (make-singer))
        (int-arrognt (make-arrogant-professor fname lname)))
    (lambda (message)
      (lambda (self . args)
        (apply delegate-to-all (list int-singer int-arrognt)
               self message args)))))

(ask zoe 'SAY '(the sky is blue))
the sky is blue tra la la
the sky is blue obviously
```

6/8

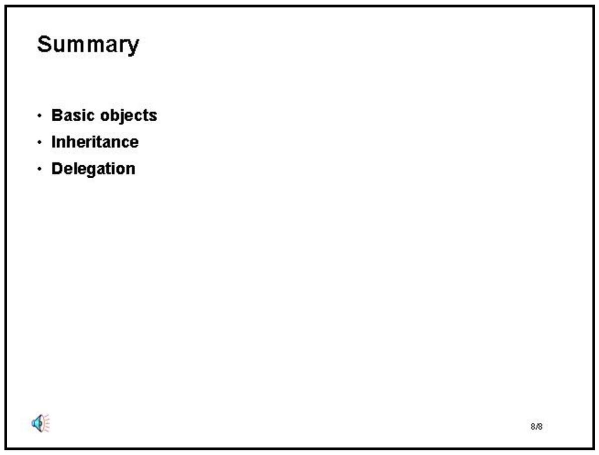
Slide 14.7.6
Clearly the order in which we list objects will determine the behavior we see. Checking the singer first, then the professor will give one kind of behavior, while checking the professor first, and then the singer will give a different kind of behavior. We could add more things to our ability to use multiple inheritance. For example, suppose we want to pass a message on to **all** the internal objects, and have them all do the appropriate thing. For example, we could have a singing arrogant professor with two internal objects, as before, but now when we ask it to do some thing, we will pass the message on to all the internal objects. Notice the different behavior we get for the examples in this case.

Slide 14.7.7
And this comes from a particular choice, as shown here in the procedure that delegates a message to all objects.


Unusual Multiple Inheritance: delegate-to-all

```
(define (delegate-to-all to-list from message . args)
  (foreach
    (lambda (to-whom)
      (apply delegate to-whom from message args))
  to-list)
```

7/8


Summary

- Basic objects
- Inheritance
- Delegation

8/8

Slide 14.7.8
So what we have shown you is how to build an object oriented system, especially considering the kinds of behaviors we can get. We saw the role of classes, instances and hierarchies of classes that capture common behavior.
Once we have the ability to create variations on objects, we have to worry about how to allocate requests for actions. We can **delegate** to particular objects. We can **inherit** from super-classes. If we have multiple inheritance, we have lots of variations in how objects inherit methods from different super-classes.
Thus we have begun to see the range of behaviors available in such systems. The goal is to try to design classes that support the desired behaviors, in a modular fashion. This includes deciding what each class should do and the interactions between the classes. If we make a poor design decision, we can get very unexpected behavior, and our goal is to guard against this.