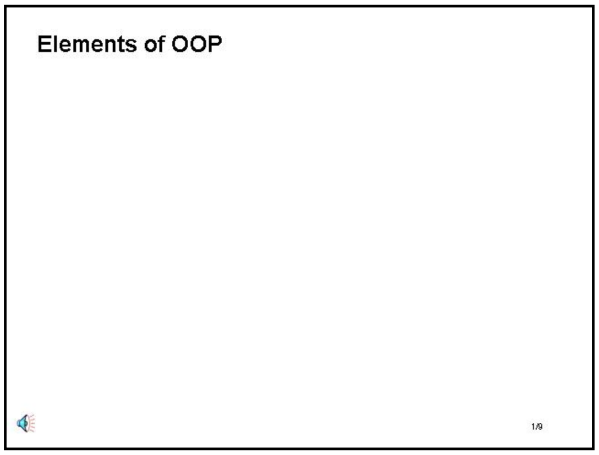
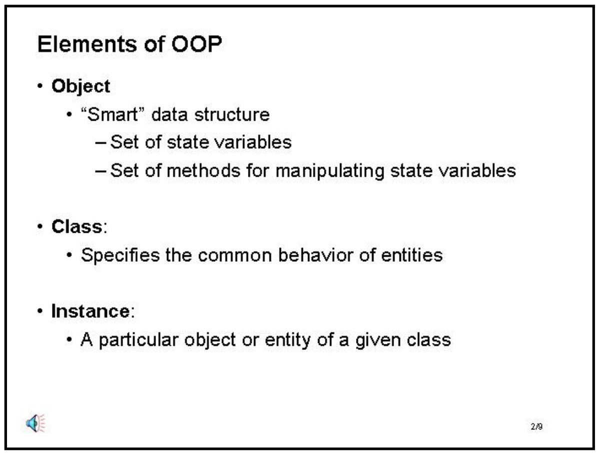
6.001 Notes: Section 14.1

Slide 14.1.1


Elements of OOP

1/9

Last time we introduced the idea of object oriented systems. Today, we want to build on those basics, looking at how we can expand object-oriented systems to deal with hierarchies of objects, to leverage the commonality of methods between different kinds of objects. We are going to see how the same ideas of abstraction allow us to control complexity in object-oriented systems, much as they did in procedural systems. We are also going to show a version of a Scheme based object-oriented system. Note that this is not an optimal implementation of an object-oriented system compared to say Java or C++, but the goal is to show how one could create and manipulate an object-oriented system using the tools we have already developed. Our intent is to separate these issues; highlighting the concepts of object-oriented systems, while grounding those concepts in a specific instantiation in Scheme.


Elements of OOP

- **Object**
  - "Smart" data structure
    - Set of state variables
    - Set of methods for manipulating state variables
- **Class**:
  - Specifies the common behavior of entities
- **Instance**:
  - A particular object or entity of a given class

2/9

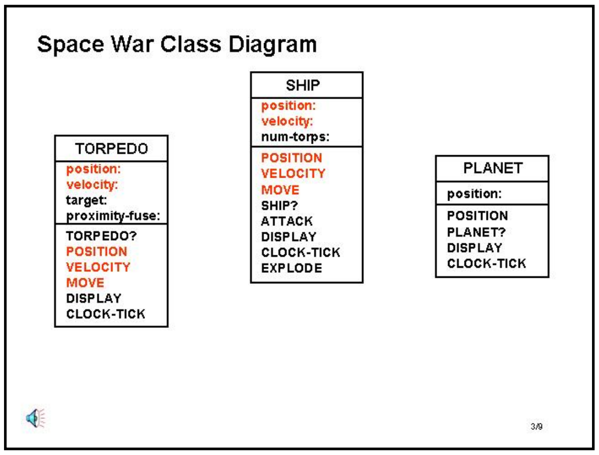
Slide 14.1.2

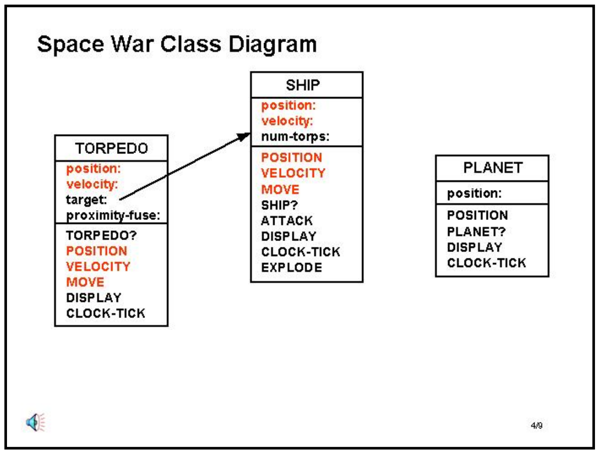
To start, let me remind you of what we have already seen. We are trying to organize a large system around a collection of objects. An object can be thought of as a "smart" data structure; a set of state variables that describe the object; and an associated set of methods for manipulating those state variables. We expect our systems to have many different versions of the same kind of object. For instance, think of bank system in which we might have many different accounts. Each account would have a set of data values: current balance, overdraft protection, pending deposits. Thus there is the notion of an account as an abstract structure, and there is notion of different, specific versions of this abstract structure. Thus, we make a distinction. A **class** will define the common behavior of a kind of object in our system, the collection of things that are going to behave in the manner defined by those commonalities. **Instances** capture the specific details of an individual version of that class.

Our goal is to see how to structure computation within this new paradigm, in which the central units are "smart" data structures. Thus, when we design a system we will tend to focus on the classes, as those are the basic building blocks of our system. The "programming" of these systems will tend to focus on the interactions and behaviors of the classes: how should changes in one instance of an object affect other aspects of that object or other objects in the system?

When we want to actually run our system, say to simulate a behavior, we will use the classes to create the instances that are particular versions of things, and then see how those specific instances interact.

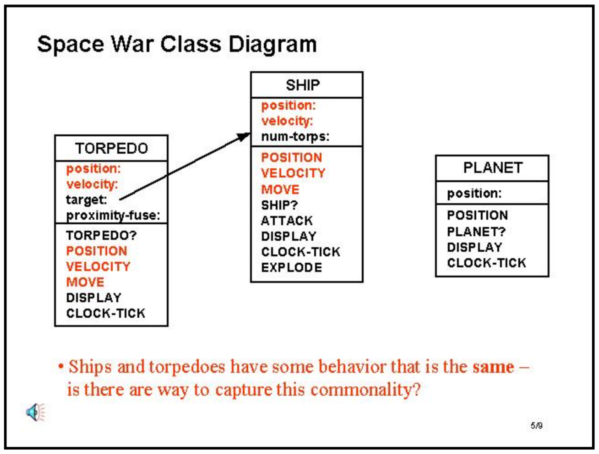
Slide 14.1.3
Recall from last time the example we used to demonstrate object-oriented systems. That example was a system for simulating a simple "star wars" scenario. In that system, we had class diagrams such as those shown here. We had a class for ships, a class for planets, a class for torpedoes, and some other classes that we have not shown here. Recall that for each class, we had two sets of things: internal state variables, which characterized the state of each instance of the class, and a set of methods, the things that the class was capable of doing. Those methods often were characterized by the instance accepting a message of the same name, then performing some interaction between different objects within the system or changing the status of the internal state variables of an instance.



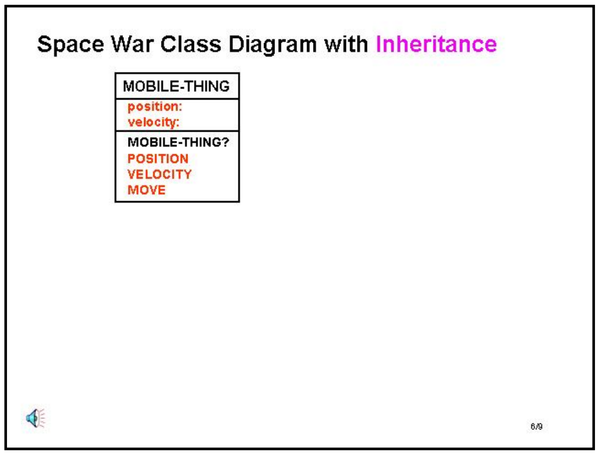


Slide 14.1.4
In order to make a point, I have changed slightly my definition of a torpedo. Last time a torpedo just used a position and a velocity, and it moved until it hit some thing, at which point it exploded. A smarter torpedo might explicitly seek some target, and use a state variable like a proximity fuse, so that when the torpedo got close enough to its target it would explode.
One reason for introducing this is to notice that state variables within our instances could actually point to other instances. So the state variable for a target would point to another class, a ship in this case.

Slide 14.1.5
However, the real point we want to draw attention to in this diagram is the commonality, particularly the commonality between ships and torpedoes. Note that both of these objects can fly, they both therefore have state information about position and velocity, they both have methods that deal with position and velocity; thus they have a lot of things in common, as well as having a few distinctive properties.
We know that a common theme in this course is capturing common patterns and abstracting them. So the issue here is whether we can do the same thing in an object-oriented system. Can we take advantage of the fact that torpedoes and ships share a lot in common, and use that to build more modular systems?



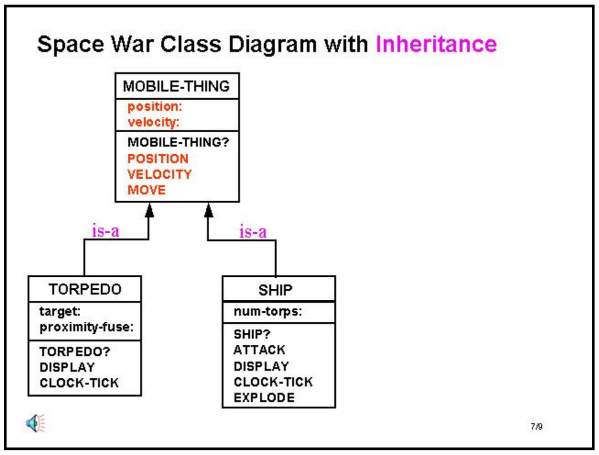
• Ships and torpedoes have some behavior that is the **same** – is there are way to capture this commonality?



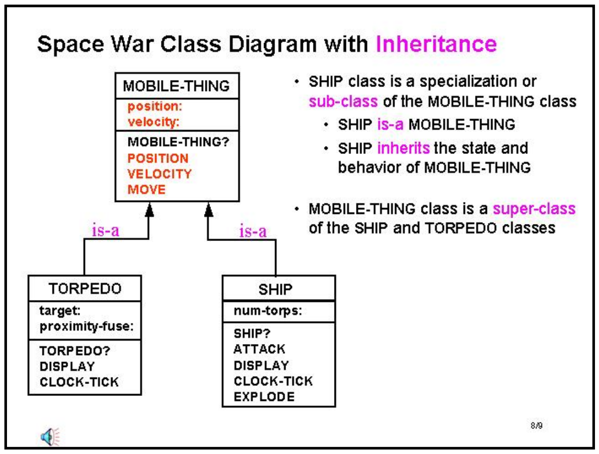
Space War Class Diagram with Inheritance

Slide 14.1.6
Conceptually we should be able to do this. Without worrying about implementation details, let's first pull out that common pattern in our class diagram. Here is an abstraction of that common pattern, a new class called a **mobile-thing**. It has two state variables, **position** and **velocity**, and it has some common methods for dealing with those variables. These variables and methods will hold for any mobile object. We will, of course, include a type tag for this new kind of class, as well. And this thus defines a new class.

Slide 14.1.7
Given that new class, we can now create specializations. We can create a **subclass**. A torpedo is a particular kind of mobile thing. It has all the properties of mobile things, but it also has characteristics that are particular to torpedoes. Similarly, a ship is a kind of mobile thing; it's a specialization that has, in addition to the properties of mobile things, other characteristics that matter only to ships.



Space War Class Diagram with Inheritance

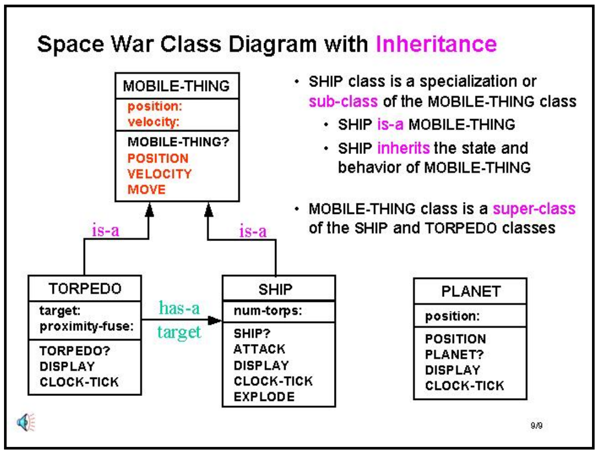


Space War Class Diagram with Inheritance

- SHIP class is a specialization or sub-class of the MOBILE-THING class
  - SHIP is-a MOBILE-THING
  - SHIP inherits the state and behavior of MOBILE-THING
- MOBILE-THING class is a super-class of the SHIP and TORPEDO classes

Slide 14.1.8
As we start designing our system, we can begin to put together hierarchies of class diagrams. We have **base classes** like mobile things. We also have some specializations or **subclasses** so that for example a ship is a mobile thing and should therefore inherit the state and behavior of a mobile thing, as well as having its own properties. In the other direction, we say that the mobile thing class is a **super-class** of the ship and torpedo classes.
Now we can start building a broader set of designs, in which we have hierarchies of information captured in different specializations of different classes of objects.

Slide 14.1.9
And as we fill out our class diagram, we will have different relationships between the classes. Torpedoes, for example, can have a class as a link: targets are always going to be elements of the ship class.
Note that one of the advantages of creating super-classes is that we can nicely isolate the code for the methods of the super-classes so that if we want to change one of those methods, we only need to worry about the implementation of the super-class, not about all of the specializations. If for example we decide to change how things move, we don't have to search for all the **move** methods in the different subclasses, we need only change the method in the super-class. This gives us a nice



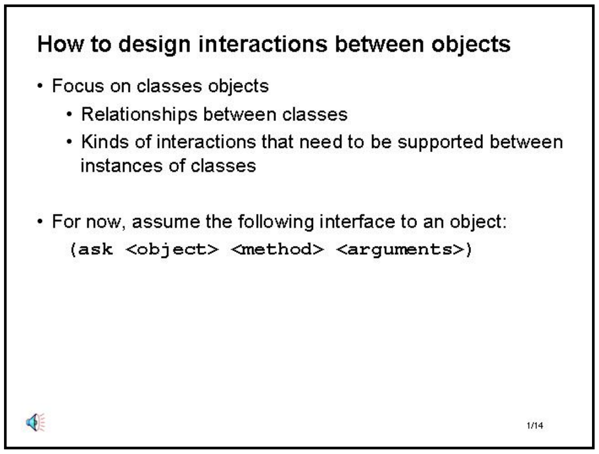
Space War Class Diagram with Inheritance

- SHIP class is a specialization or sub-class of the MOBILE-THING class
  - SHIP is-a MOBILE-THING
  - SHIP inherits the state and behavior of MOBILE-THING
- MOBILE-THING class is a super-class of the SHIP and TORPEDO classes

modularization of the system, by isolating the common methods in a single place, for easy maintenance and change.

6.001 Notes: Section 14.2

Slide 14.2.1

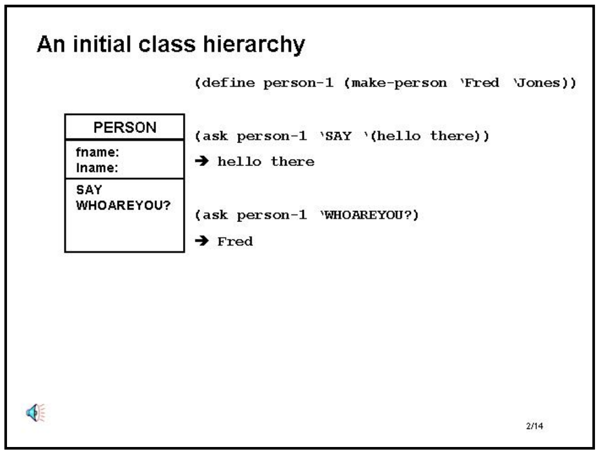

How to design interactions between objects

- Focus on classes objects
  - Relationships between classes
  - Kinds of interactions that need to be supported between instances of classes
- For now, assume the following interface to an object:
  `(ask <object> <method> <arguments>)`

1/14

So our first goal is to examine the paradigm of constructing systems around objects, and for now we are going to ignore the issue of how to describe these object systems in Scheme; we will instead initially focus on the abstract use of object-based systems. Thus, we will primarily consider the classes of objects we might want in a system, and the range of interactions that we will need to support between objects. For now, we will simply assume that given an instance of a class, that is an object, we can send that object a message using the form `(ask <object> <method> <arguments>).` In other words, we can use some Scheme interface (which we will define later) based on the `ask` procedure, which takes as arguments an object, a name of a method (something we want the object to do), and some set of arguments that specify details of the method, and `ask` will cause the object to execute the specified method. We will return to details of this shortly.

Slide 14.2.2


An initial class hierarchy

```
(define person-1 (make-person 'Fred 'Jones))

(ask person-1 'SAY '(hello there))
➔ hello there

(ask person-1 'WHOAREYOU?)
➔ Fred
```

2/14

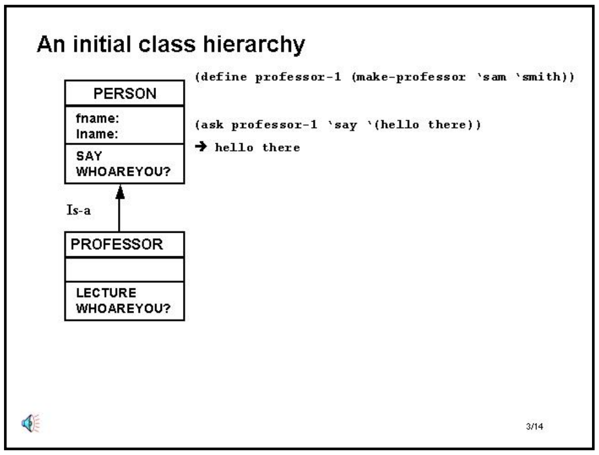
We start with a class for people, which we call the **person** class. Each instance of a person has two class variables, holding the person’s first and last name. Each instance also has two methods. The **say** method causes the person to say something. The **whoareyou?** method causes the person to indicate their first name.
The class diagram for this class is shown in the slide. As well, we will assume some constructor for making instances of the class (we will get to details of this shortly), and we can see how **ask**ing an instance of this class to say something or to identify themselves causes actions in the system.

Slide 14.2.3

Now we add a sub-class of **person,** called a **professor.**

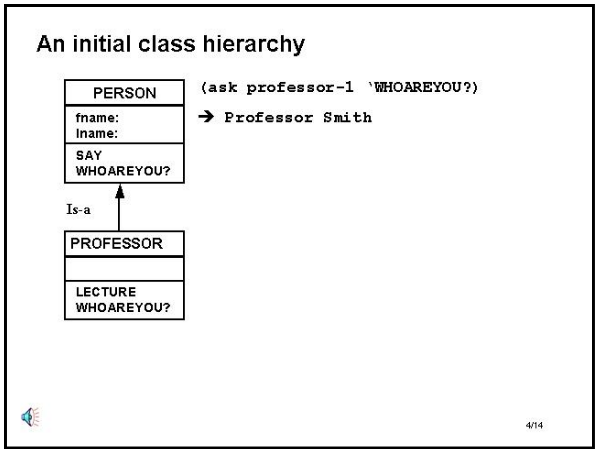
Note that this class does not have any specific internal class variables. However, because it is a subclass of a person, it should **inherit** the class variables of its **superclass.** In other words, professors also have a first and last name, because their **person** superclass instance has such variables. And professors have the ability to **say** things by virtue of being a subclass of **person.**

Note that we again assume there is some constructor for making instances of a class (which should, as we will see, take care of creating superclass instances as part of the process). Because of this hierarchy of classes, we should be able to ask the professor to say something, and it will use its inherited method from the person instance to do this.



An initial class hierarchy

```
(define professor-1 (make-professor 'sam 'smith))

(ask professor-1 'say '(hello there))
➔ hello there
```



An initial class hierarchy

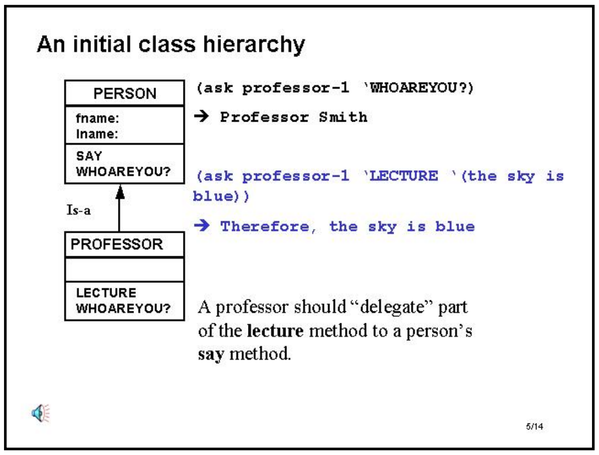
```
(ask professor-1 'WHOAREYOU?)
➔ Professor Smith
```

Slide 14.2.4

In our little world, **professors** have no class variables of their own (ah, the irony!) but they do have two methods, a **whoareyou?** method and a **lecture** method. Notice that a **professor** has its own **whoareyou?** method, distinct from the identically named method in **person.** If we ask a **professor whoareyou?** it will run its own method to answer the question, with a different behavior. When a subclass has a method of the same name as a superclass, the subclass method is said to shadow the inherited method in the superclass instance.

Slide 14.2.5

Now in the world we are creating, it is traditional that when a **professor lecture**s he starts every sentence with "Therefore". An interesting question to consider when actually implementing the **professor** class is whether this **lecture** method is a distinct method, or whether it shares structure with the underlying **say** method of the inherited **person** class. Conceptually, we would like to think that **lecturing** is a particular variant on **saying:** indeed one simply **says** the word "Therefore" and then **says** the remaining text. This idea of using a superclass' method to accomplish part of a method is called "delegation".



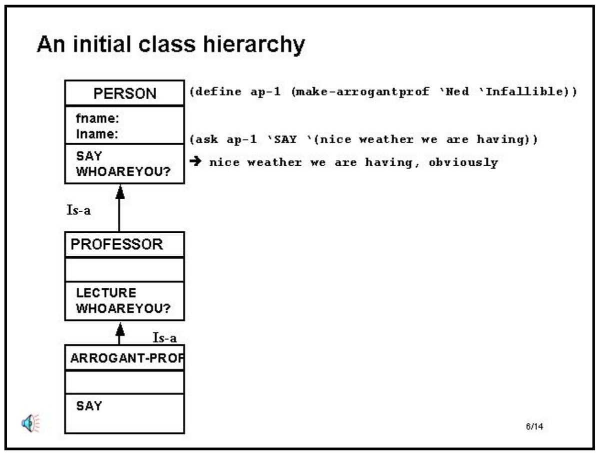
An initial class hierarchy

```
(ask professor-1 'WHOAREYOU?)
➔ Professor Smith

(ask professor-1 'LECTURE '(the sky is
blue))
➔ Therefore, the sky is blue
```

A professor should "delegate" part of the **lecture** method to a person's **say** method.

Note that this is an important requirement to place on an object system. The idea is that at a conceptual level, just as classes can be related to one another (e.g. via the subclass hierarchy), so too can methods be related to one another, by this delegation idea. And at the implementation level, delegation can be seen as a mechanism that allows subclasses to specialize (and use) methods found in superclasses.

This has two important consequences. The first is that if we design our object system correctly, we will have a clean modularity of code, so that there is only one place to implement **say**ing some thing, and thus only one place to worry about if we decide to change the manner in which this method executes. Secondly, we have an explicit indication (through the act of delegation) that the **lecture** method and the **say** method are related conceptually. We will return to this point when we consider an explicit implementation of an object-oriented system in Scheme.



An initial class hierarchy

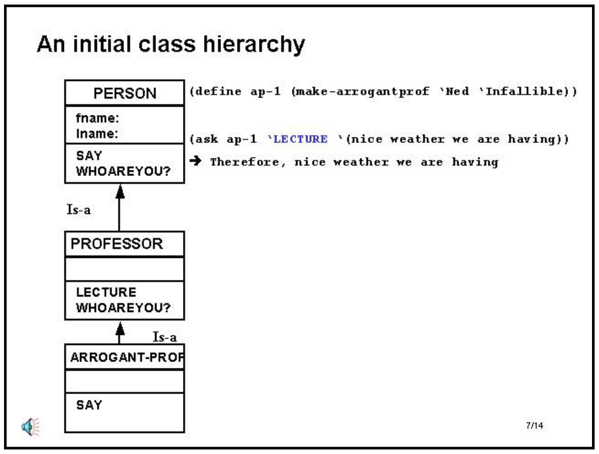
```
(define ap-1 (make-arrogantprof 'Ned 'Infallible))

(ask ap-1 'SAY '(nice weather we are having))
➔ nice weather we are having, obviously
```

Slide 14.2.6

Now, let's add another new class as a subclass of **professor,** this one called an **arrogant-prof.** An **arrogant-prof** is distinguished by the fact that he ends everything he **says** with "obviously". The obvious (sorry!) way to do this is to have the **arrogant-prof'**s method of **say**ing simply be a delegation of **say**ing to its superclass (**professor)**, with obviously tacked onto the end. Note that we delegate one step up the superclass chain because that is the only way **arrogant-prof** can eventually "get to" the **person.** The **arrogant-prof** has a pointer to its immediate superclass, but not to superclasses higher in the chain. By delegating one step up the chain, the ordinary inheritance mechanism will take over when the system determines that the **professor** doesn't have a **say** method of its own.

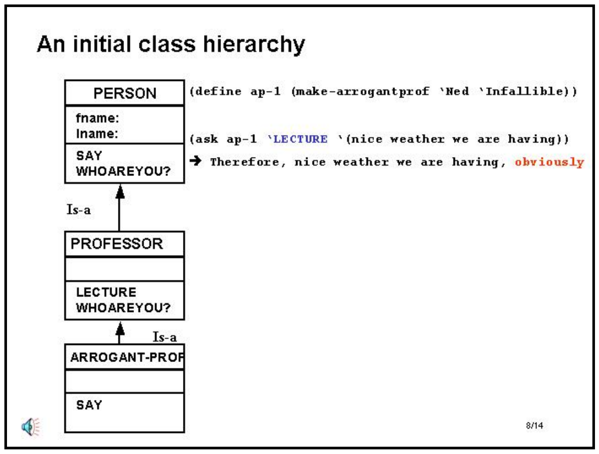
Slide 14.2.7

Now an interesting issue arises if we ask an **arrogant-prof** to **lecture.** What should he say? One view is that, because **arrogant-prof** has no **lecture** method, **lecture**'ing will be handled by the method in the **professor** class, which in turn will delegate to **person** after tacking on a "therefore". The result will be as shown.



An initial class hierarchy

```
(define ap-1 (make-arrogantprof 'Ned 'Infallible))

(ask ap-1 'LECTURE '(nice weather we are having))
➔ Therefore, nice weather we are having
```



An initial class hierarchy

```
(define ap-1 (make-arrogantprof 'Ned 'Infallible))

(ask ap-1 'LECTURE '(nice weather we are having))
➔ Therefore, nice weather we are having, obviously
```

Slide 14.2.8

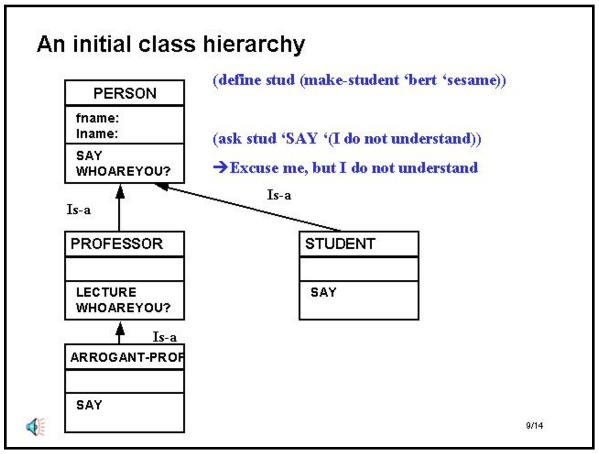
But another view says that **arrogant-prof** inherits the **lecture** method from **professor**, so conceptually the **lecture** method is sitting there in the **arrogant-prof** class. As a result, the **lecture** method should give **arrogant-prof** a chance to see whether *it* knows how to **say** something, rather than instantly delegating to a **say** method found up the superclass chain. And of course, **arrogant-prof** does know how to say something; it has a **say** method, one that should shadow the **say** method of a person. As a result, asking an **arrogant-prof** to **lecture**, under this view, should result in the following sequence of events:

- can't find a **lecture** method in **arrogant-prof** class; use the method from the superclass, **professor**
- the **lecture** method from professor asks the *current instance* (an **arrogant-prof**) to **say** the original stuff, with "therefore" prepended to the front. Note what has changed and what hasn't under this view: **Lecture**ing is still **say**ing something with "therefore" on the front, but now we are asking the current instance, an **arrogant-prof**, to do the **say**ing, instead of handing it up the superclass chain.
- the **arrogant-prof** has a **say** method, which works by adding "obviously" to the end of the statement, and finally, delegating the resulting statement up the inheritance chain to the next **say** method that can be found (which turns out to be in the **person** class).

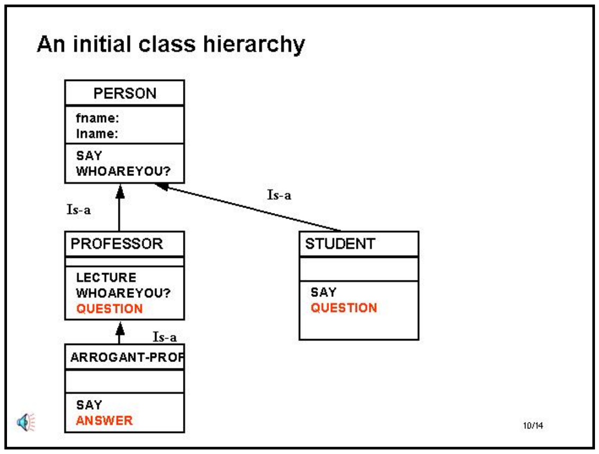
The result is as shown.

A key issue, we will then see, is to decide as we build our object oriented system, how we want methods to inherit from other methods. Ideally we will want the ability to choose as a programmer, when we decide on our classes of objects, whether we want to delegate behaviors to specific superclasses, or to simply inherit behaviors by following up the chain of classes.

Slide 14.2.9
Now let's add one final type of object to our system, a **student. Students** are always very polite, so they have their own **say** method, which prepends the words "Excuse me, but" to the front of everything they say.

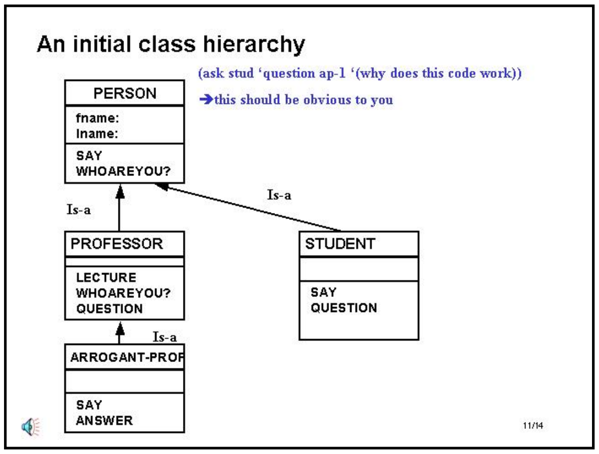

An initial class hierarchy

```
(define stud (make-student 'bert 'sesame))

(ask stud 'SAY '(I do not understand))
➔Excuse me, but I do not understand
```

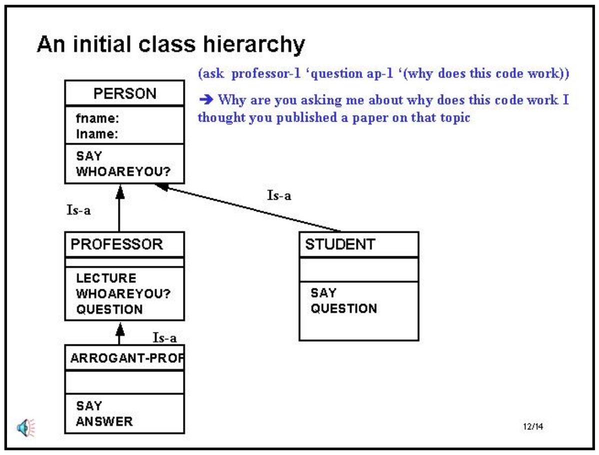

An initial class hierarchy

Slide 14.2.10
Now suppose we go back to our class diagram, and decide to modify our classes. For example, we want **students** to have the ability to ask a **question** of a professor. And of course, we need to add the ability for a professor to **answer,** which we choose to do within the **arrogant-prof** class. But we also decide that occasionally even a **professor** might have the need to ask a question.
In terms of our object system, this simply would require redefining whatever mechanism we use to create classes and instances, so that a new method is included in that class definition.

Slide 14.2.11
Note that in this case, we want our objects to take several arguments, in particular, both the question and the object to which the question is being directed. Also note the use of `ap-1` as an argument, meaning we want the actual instance, not just the name of that instance.
We still have to define how we want the **arrogant-prof** to respond. We choose to incorporate the following behavior. If the question is being asked by a student, then the **arrogant-prof** will respond by **say**ing "this should be obvious to you".


An initial class hierarchy

```
(ask stud 'question ap-1 '(why does this code work))
➔this should be obvious to you
```



An initial class hierarchy

```
(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
```

Slide 14.2.12

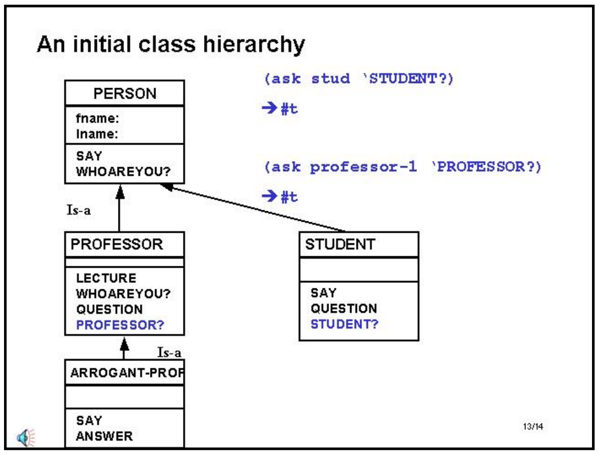
On the other hand, if the question is asked by another **professor,** then the **arrogant-prof** will respond by **say**ing "why are you asking me about … I thought you published a paper on that topic"

Slide 14.2.13

The reason for introducing this behavior is to provide an example of a class method, in which the action of the method depends on the object that initiated it. In particular, to incorporate this behavior, the **arrogant-prof'**s **answer** method will need to do something different depending on what kind of object ask the **question** in the first place: if it was a **student** then we want to respond one way, whereas if it was a **professor,** we want to respond a different way.

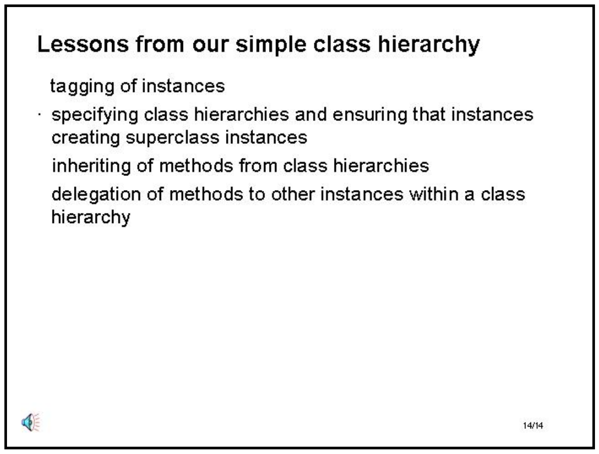
How do we do this? In essence, we need a way of "tagging" our objects, to identify their type. Within an object-oriented system, we choose to do this by adding to our classes a "predicate" method, which responds to the question of whether the instance is of that specified type. Thus, we can ask an object if they are a student, or a professor.

(Actually, if you think about this, you will realize that we need to be careful in how we do this, since asking a **professor** if he is a **student** could lead to an error if the fact that the **professor** object does not have a **student?** method is not handled properly. We will see a particular way of dealing with this issue later.)



An initial class hierarchy

```
(ask stud 'STUDENT?)
➔#t

(ask professor-1 'PROFESSOR?)
➔#t
```



Lessons from our simple class hierarchy

- tagging of instances
- specifying class hierarchies and ensuring that instances creating superclass instances
- inheriting of methods from class hierarchies
- delegation of methods to other instances within a class hierarchy

14/14

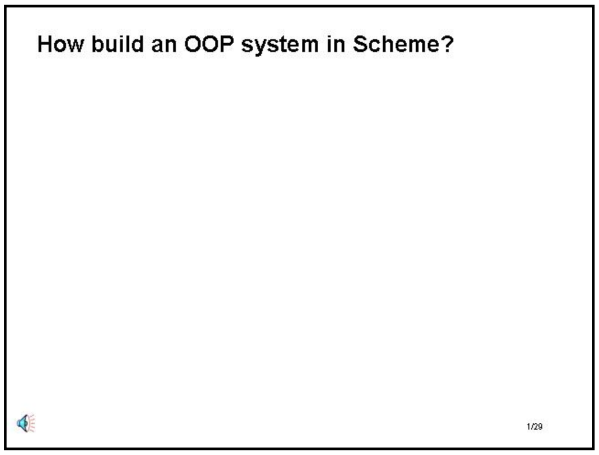
Slide 14.2.14

So what are the lessons we can take away from this simple design exercise? Well, we can see that as we design a system for supporting the creation of object-oriented systems, we need methods for dealing with:

- tagging of instances
- specifying class hierarchies and ensuring that instances create superclass instances
- inheriting of methods from class hierarchies
- delegation of methods to other instances within a class hierarchy

We will return to these issues in detail in later parts of the lectures.

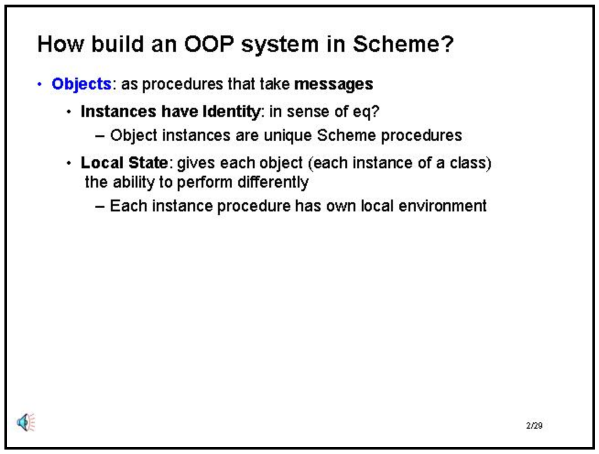
6.001 Notes: Section 14.3



How build an OOP system in Scheme?

1/29

Slide 14.3.1
Now, let's revisit these ideas by looking at how one might build an actual object-oriented system in Scheme. We are going to intertwine the idea of building class systems in Scheme, with the idea of using classes as a design methodology. Although not optimal, we will use a language we understand (Scheme) as a basis for examining how to create procedures that support class based systems, while trying to abstract the general methods from the specifics of this implementation in Scheme.

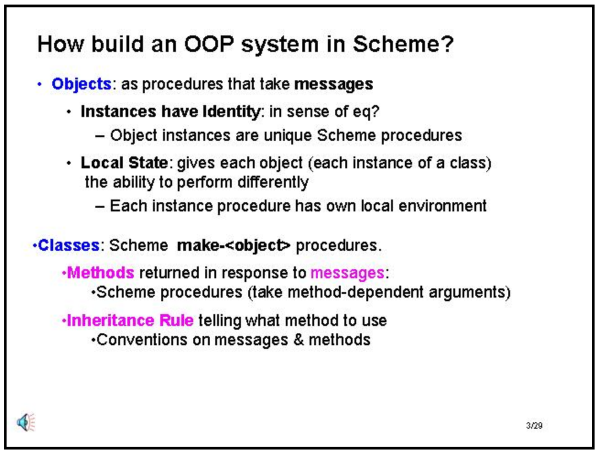


How build an OOP system in Scheme?

- **Objects**: as procedures that take **messages**
  - **Instances have Identity**: in sense of eq?
    - Object instances are unique Scheme procedures
  - **Local State**: gives each object (each instance of a class) the ability to perform differently
    - Each instance procedure has own local environment

2/29

Slide 14.3.2
Here is our blueprint for making this happen. First of all, objects in our system, i.e. the particular instances, we will represent as procedures that accept messages as input. We will use the same idea we saw earlier, message-passing procedures that accept messages and cause changes to state variables as a function of those messages. One of the advantages of this choice is that each instance will be uniquely identified. We can use `eq?` to tell them apart, because each object instance will be a unique Scheme procedure with a local frame that captures its information as local state.
Each object or instance can be formed differently because it has a procedure that points to a local environment that captures the state information, just as we saw in the previous examples of the earlier lectures.
Thus we will represent instances as local procedures with local state captured in local environments.

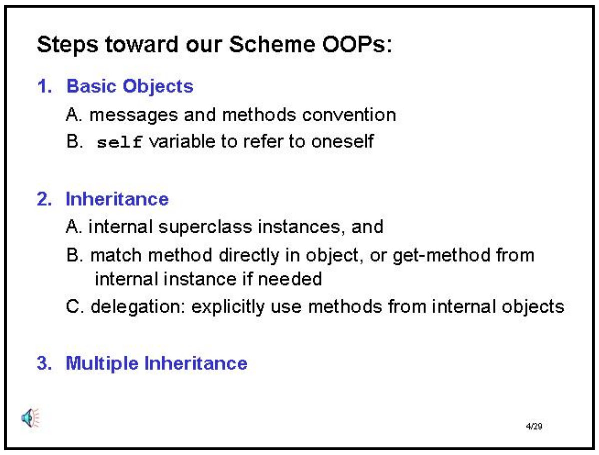


How build an OOP system in Scheme?

- **Objects**: as procedures that take **messages**
  - **Instances have Identity**: in sense of eq?
    - Object instances are unique Scheme procedures
  - **Local State**: gives each object (each instance of a class) the ability to perform differently
    - Each instance procedure has own local environment
- **Classes**: Scheme **make-<object>** procedures.
  - **Methods** returned in response to **messages**:
    - Scheme procedures (take method-dependent arguments)
  - **Inheritance Rule** telling what method to use
    - Conventions on messages & methods

3/29

Slide 14.3.3
Instances, of course, are simply particular versions of a class. So we will also need in our system a way of defining classes, and we will use a particular convention. We will define a Scheme procedure called `make-`something (e.g. `make-person`, `make-professor`). Inside of that procedure, that definition of a class, we need two things: we will need a set of methods that will be returned in response to messages, so the class procedures will need to contain within themselves ways of taking in a message and returning a local method. The second thing we need is an inheritance chain, a way of telling the class what methods to use, and by this we mean not just the local method, but rather if the class is a subclass of some superclass, we need a convention for deciding how to inherit methods from that superclass. In the case of a professor, for example, we might have methods to deal with LECTURE or WHOAREYOU?, but we also want to inherit the method of SAY from the superclass of PERSON. So we will need to set up conventions for inheriting methods from 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**

4/29

Slide 14.3.4

So here is the blueprint we are going to use throughout the rest of this lecture. We will start by building basic objects, determining the methods we need and the conventions for dealing with such methods. We will see why we need a particular way of getting a handle on the object itself as an argument to methods, as we will see in a set of examples.

We will then turn to the notion of inheritance, how to build an internal superclass, and then deal with getting methods from that object itself or from internal instances of superclasses. We will see that there are variations on inheritance that become important when designing object-oriented systems.

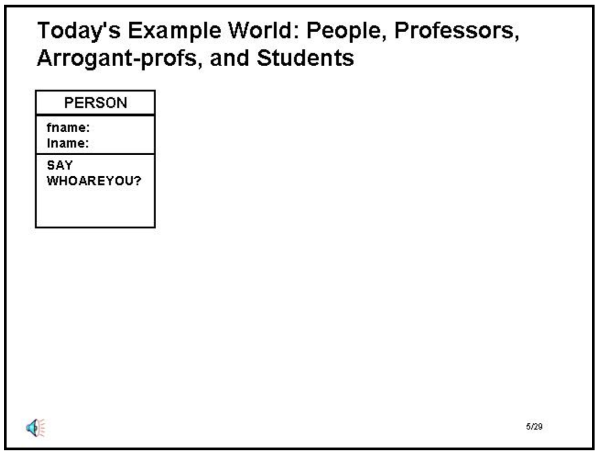
Finally, we will turn to the notion of multiple inheritance, having objects that can inherit methods from different kinds of superclasses.

So we are going to use this blueprint both to build an object oriented system in Scheme, and to see how such ideas can be used in general to design systems around the concepts of objects as a tool for controlling complexity in large systems.

Slide 14.3.5

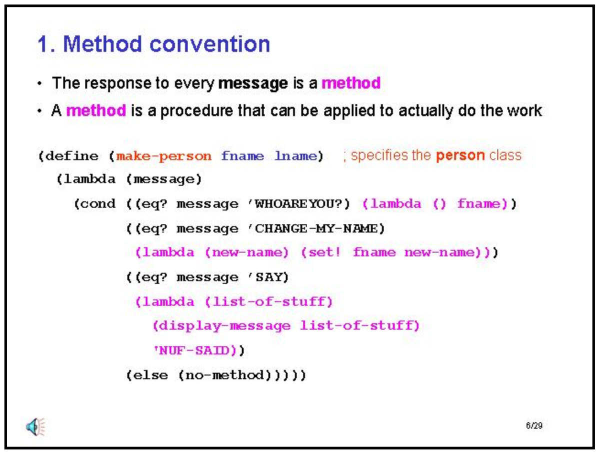
We are going to develop these ideas using the particular example we developed previously, with a world of **people, professors, arrogant-profs** and **students**. As we progress, we will see how these classes relate to one another in terms of superclasses and inheritance of methods.

To start, we design a first class. This means creating a class diagram, and we will start with the simplest thing: a **person**. Here is the definition for the person class. It has two internal variables: an `fname` and an `lname`. It also has some methods that it can handle: returning its name, and saying things (since that is what a person "does").



Today's Example World: People, Professors, Arrogant-profs, and Students

5/29



1. Method convention

- The response to every **message** is a **method**
- A **method** is a procedure that can be applied to actually do the work

```
(define (make-person fname lname)   ; specifies the person class
  (lambda (message)
    (cond ((eq? message 'WHOAREYOU?) (lambda () fname))
          ((eq? message 'CHANGE-MY-NAME)
           (lambda (new-name) (set! fname new-name)))
          ((eq? message 'SAY)
           (lambda (list-of-stuff)
             (display-message list-of-stuff)
             'NUF-SAID))
          (else (no-method)))))
```

6/29

Slide 14.3.6

Now we are ready to implement our first class. Remember we said our class will be characterized as a procedure which, when invoked, will construct instances of the class. Here is our `make-person` procedure, which defines our **person** class.

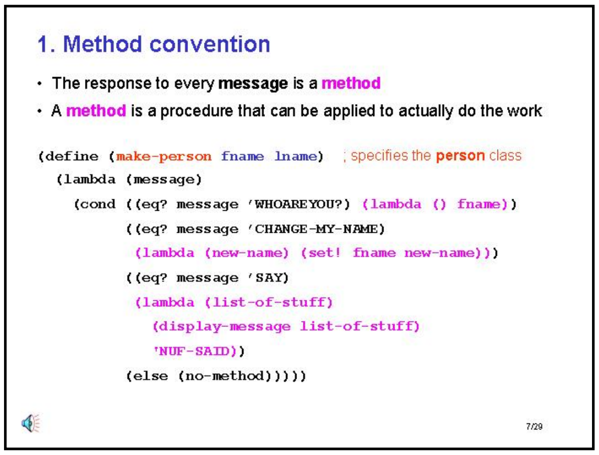
To set this up, we have to make some design choices. Remember that instances of a class do different kinds of things. It may simply want to return some information, e.g. its name. It may want to change internal information, e.g. change the name. Thus we need methods that return information and methods that change information, and here is our design choice. We are going to require the response of an instance to every message to be a **method**. This is simply a choice on our part, though we believe it will be a convenient one as we add capabilities to our system. Note that it says we can't just return a value; we return procedures, the application of which may then return a value.

Slide 14.3.7

So let's look at this definition to see how this is implemented. Our class definition, `make-person`, takes as input an fname and an lname. It returns a message-accepting procedure, which will constitute one of our instances. Inside of that `lambda` we have several things.

We have a set of dispatches, based on the message. Thus, if the message is `WHOAREYOU?,` we will return a procedure of no arguments, which when evaluated will return the value of the name. We have a way of changing the name: again it is a procedure, now of a single argument, which will change the binding for `fname` within the local environment to the new value, using the same methods we saw before. And we will have a way of "saying" something, and in this case, when asked to `SAY` something, we will return a procedure that takes as input a list of things, and displays that as output on the monitor.

Note as our default clause, if the message is not recognized, we will deal with that by a special procedure designed to handle the case of a missing method (which we will return to later).

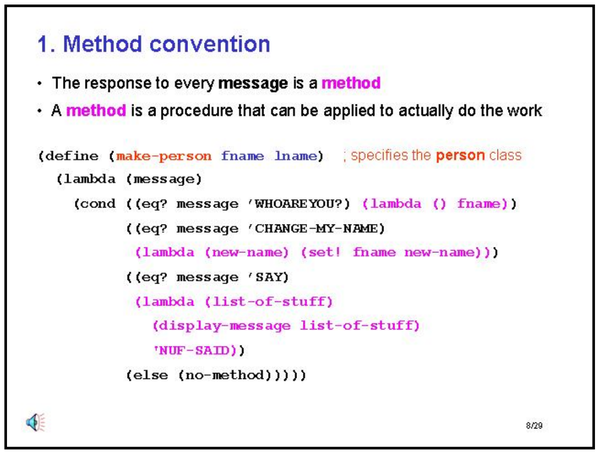


1. Method convention

- The response to every **message** is a **method**
- A **method** is a procedure that can be applied to actually do the work

```
(define (make-person fname lname)   ; specifies the person class
  (lambda (message)
    (cond ((eq? message 'WHOAREYOU?) (lambda () fname))
          ((eq? message 'CHANGE-MY-NAME)
           (lambda (new-name) (set! fname new-name)))
          ((eq? message 'SAY)
           (lambda (list-of-stuff)
             (display-message list-of-stuff)
             'NUF-SAID))
          (else (no-method)))))
```

7/29



1. Method convention

- The response to every **message** is a **method**
- A **method** is a procedure that can be applied to actually do the work

```
(define (make-person fname lname)   ; specifies the person class
  (lambda (message)
    (cond ((eq? message 'WHOAREYOU?) (lambda () fname))
          ((eq? message 'CHANGE-MY-NAME)
           (lambda (new-name) (set! fname new-name)))
          ((eq? message 'SAY)
           (lambda (list-of-stuff)
             (display-message list-of-stuff)
             'NUF-SAID))
          (else (no-method)))))
```

8/29

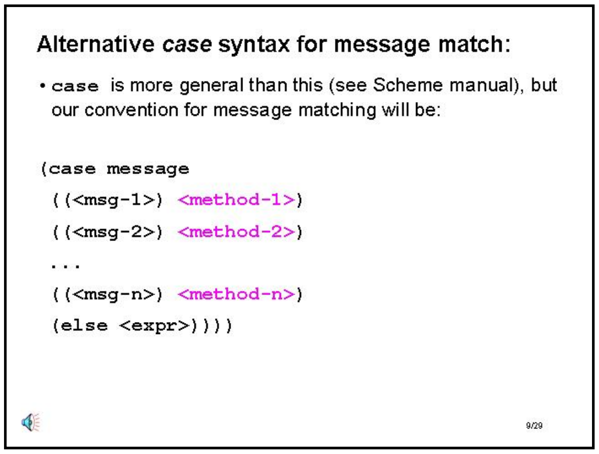
Slide 14.3.8

Notice again the general form of our system. For any message passed in as input, our class instance will return a procedure, a method for handling information. Thus we are guaranteed that the returned value is always a procedure. To see that, notice that the response to every message is a `lambda` expression.

Slide 14.3.9

We can see that the kinds of objects we will be building are going to essentially be "dispatch on message" procedures. To take advantage of this, we have a cleaner way of coding such procedures. This uses an alternative syntax called `case`. For our purposes, a `case` expression takes a single expression, called the `message` and a series of clauses. Evaluation of a `case` expression is to match the value of `message` sequentially against the first expression of each of the clauses. If a match is found, then the value of the rest of that clause is returned as the value of the entire `case` expression, in this case giving us a procedure as our method. If we do not find a match, then as in a `cond` expression, the `else` clause will always match, and we return the value of its subsequent expression.

This simply gives us a cleaner template by abstracting away the dispatch on type components of the code.

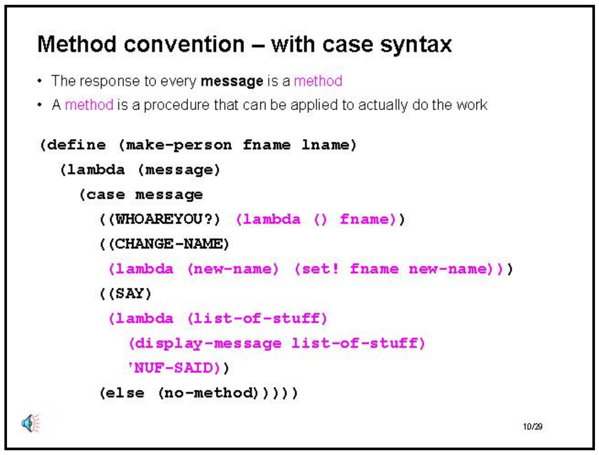


Alternative *case* syntax for message match:

- `case` is more general than this (see Scheme manual), but our convention for message matching will be:

```
(case message
  ((<msg-1>) <method-1>)
  ((<msg-2>) <method-2>)
  ...
  ((<msg-n>) <method-n>)
  (else <expr>))))
```

9/29



Method convention – with case syntax

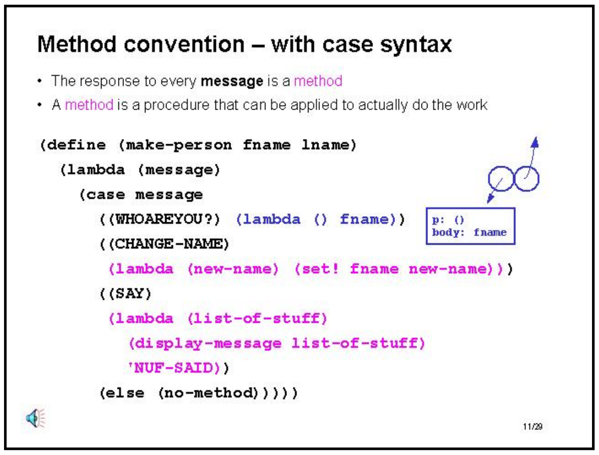
- The response to every **message** is a method
- A method is a procedure that can be applied to actually do the work

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

10/29

Slide 14.3.10
Returning to our `person` class definition, we can make our implementation more transparent, with exactly the same behavior as the earlier version.

Slide 14.3.11
Notice what is returned in each case. If a person (an instance created by this procedure) is given the message `WHOAREYOU?,` it will evaluate the subsequent clause of the first case, and return a procedure of no parameters, and a body that is just the expression `fname`. But remember where the environment pointer of this procedure will point: an environment that captures the information about the binding of `fname`, so that we will be able to return that value.

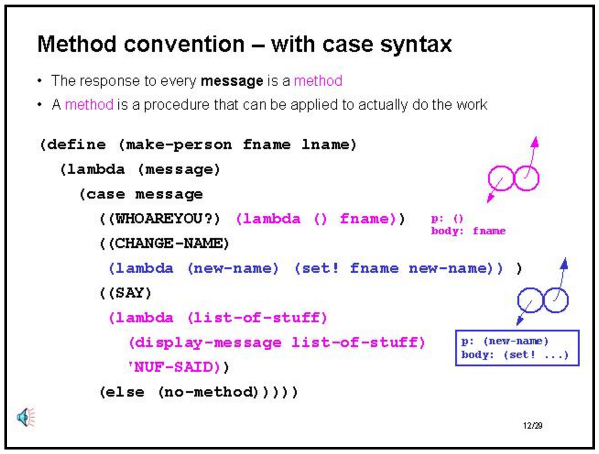


Method convention – with case syntax

- The response to every **message** is a method
- A method is a procedure that can be applied to actually do the work

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

11/29



Method convention – with case syntax

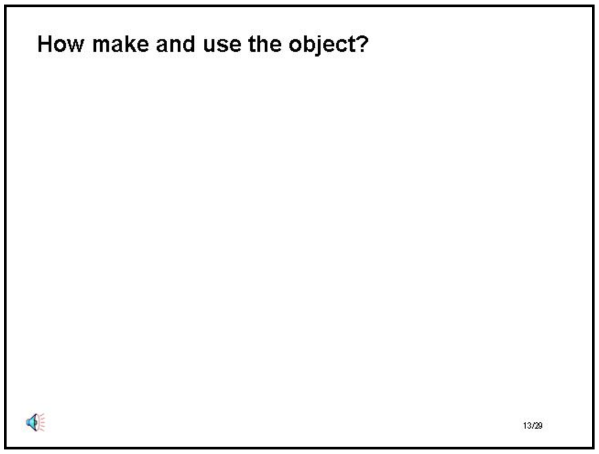
- The response to every **message** is a method
- A method is a procedure that can be applied to actually do the work

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

12/29

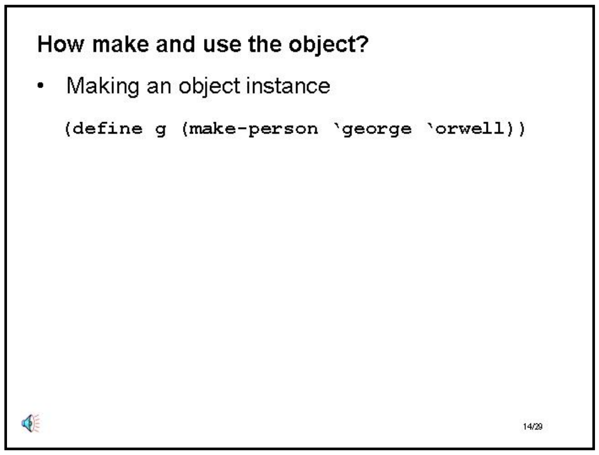
Slide 14.3.12
A similar situation holds for the message `CHANGE-NAME`, returning a different procedure, which in this case when evaluated will change the binding for `fname`.

Slide 14.3.13
Notice what we have done. We have created a first definition of a class. By defining `make-person` we have laid out a first pass at the internal variables and methods that each instance of a person will need.



How make and use the object?

13/29



How make and use the object?

- Making an object instance

```
(define g (make-person `george `orwell))
```

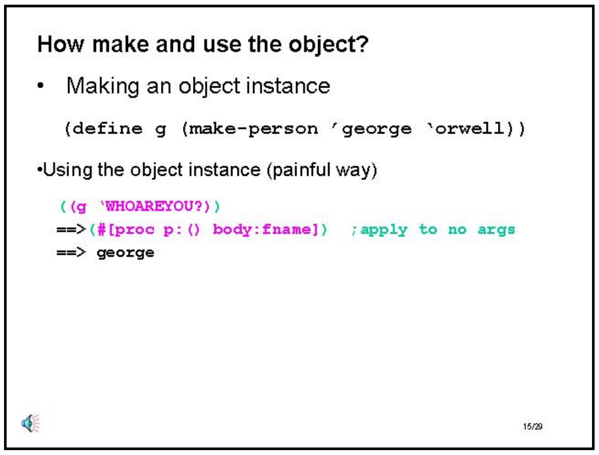
14/29

Slide 14.3.14

Now what about our distinction between classes and instances? A class defines a general set of objects. To actually use it, we need to make specific versions of members of that class, we need to create instances. To do that, we simply invoke our `make` procedure with appropriate arguments. Here is an example of a person, called George, though notice that the name of the object in the environment is just `g`. Thus `g` points to the actual person object, that has within it information about what its name is.

Slide 14.3.15

Great, we now have an instance in our system. How do we get this instance to tell us its name?



How make and use the object?

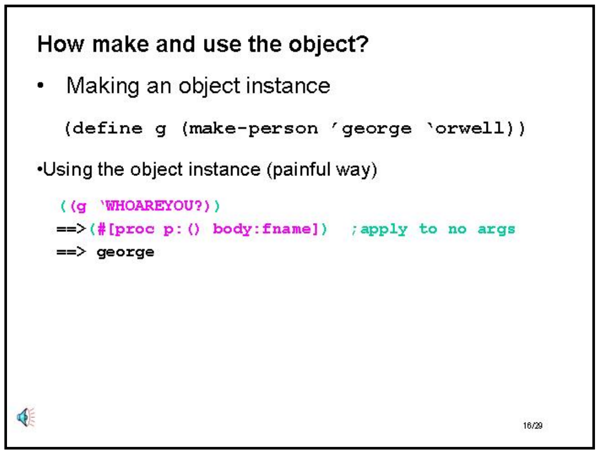
- Making an object instance

```
(define g (make-person 'george `orwell))
```

•Using the object instance (painful way)

```
((g `WHOAREYOU?))
==>(#[proc p:() body:fname])   ;apply to no args
==> george
```

15/29



How make and use the object?

- Making an object instance

```
(define g (make-person 'george `orwell))
```

•Using the object instance (painful way)

```
((g `WHOAREYOU?))
==>(#[proc p:() body:fname])   ;apply to no args
==> george
```

16/29

Slide 14.3.16

Recall that in our particular version of the system, objects are represented as procedures so we need to pass that object a message, e.g. we need to give it the message `NAME`. Remember from the previous slide what this will do. g can be applied to the argument `WHOAREYOU?` and that will return another procedure, of no arguments, whose body is just the expression `fname`. This is the **method** that is returned to us.

But remember that in our choice of implementation this method is a procedure, so we need to apply it to get it to execute its actions. In this case, we apply it to no arguments, which will cause the procedure to lookup the value of the symbol `fname` in the local environment and return it. Convince yourself that this is correct by tracing through the environment model.

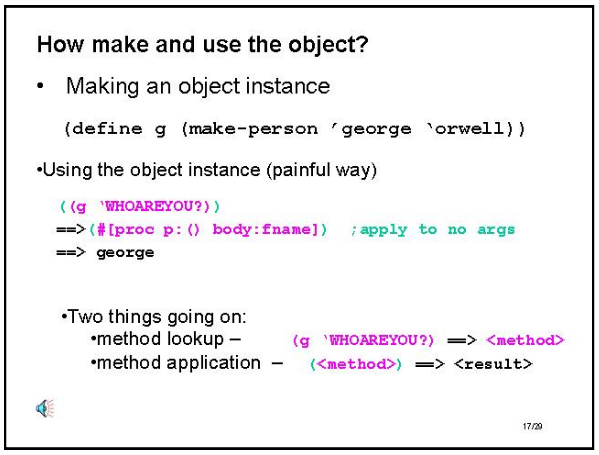
Slide 14.3.17

But this is pretty ugly, right? In order to get an object to do something, we have to do this strange invocation, `((g 'WHOAREYOU?))` with two parentheses on either end.

Why?

Because we have really intertwined two things together in our current approach. We have the process of getting the method from the object, by sending the object a message. That is the part in purple. That is the part that creates the procedure representing our actual method. Then we have a part that uses that method to make the action take place, where we take that method and apply it (since it is a procedure) to get the effect or result we want.

So we have really coupled two things together here and this leads to a messy interface as a consequence, since the user has to remember how to call things in this particular way. It would be much better if we could separate out these two actions.



How make and use the object?

- Making an object instance

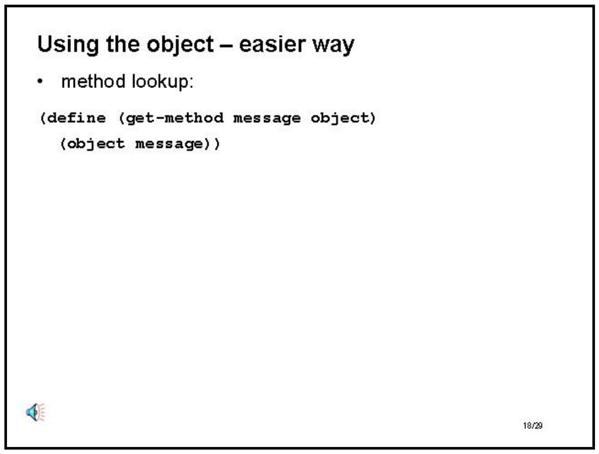
```
(define g (make-person 'george 'orwell))
```

•Using the object instance (painful way)

```
((g 'WHOAREYOU?))
==>(#[proc p:() body:fname])   ;apply to no args
==> george
```

•Two things going on:
•method lookup – `(g 'WHOAREYOU?) ==> <method>`
•method application – `(<method>) ==> <result>`

17/29



Using the object – easier way

- method lookup:

```
(define (get-method message object)
  (object message))
```

18/29

Slide 14.3.18

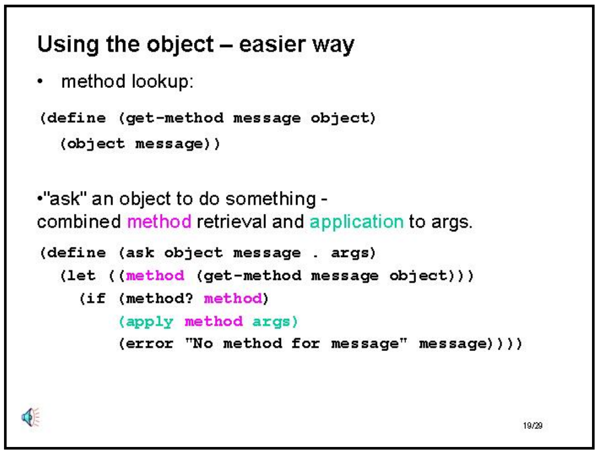
In fact, good design suggests that we should separate out these two pieces. We don't want to couple together two things like this, when we may want to separately extract methods and use methods.

First we will create a procedure for **getting** a method. Note that is just sends a message, and by convention we get back some internal procedure that will point to frames that are scoped by the actual object's frame, and thus will have access to internal variables. This builds a clean interface to our convention, by allowing us to send any message to any instance, and get back the associated method for meeting that message.

Slide 14.3.19

Separating out the idea of getting a method for an object allows us to use the returned method in a variety of ways. In some cases, we will want to pass it directly on to other objects, in other cases we will want to use it immediately. And this leads to the second part.

We also want to **ask** an object to do something. In this case we will combine our method retrieval part with the actual invocation of that method (the application of that method to a set of arguments). So here is a generic interface to do that. Note the form of this procedure, `ask`. It takes an object and a message and gets the method associated with that message. Actually, we try to get that method, but we carefully check to ensure there is an actual method available. If there is a method for this message, we then apply it to the arguments.



Using the object – easier way

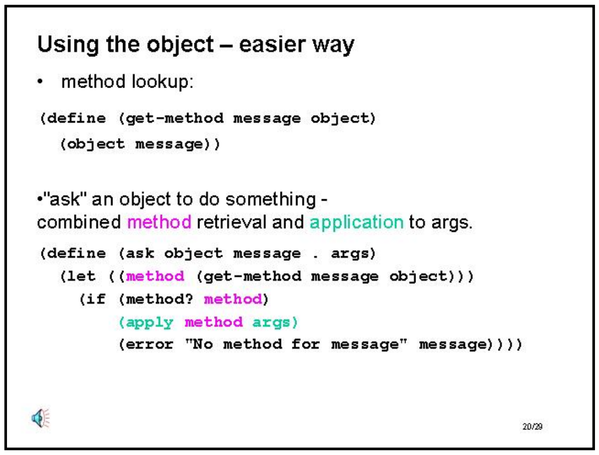
- method lookup:

```
(define (get-method message object)
  (object message))
```

•"ask" an object to do something - combined method retrieval and application to args.

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

19/29



Using the object – easier way

- method lookup:

```
(define (get-method message object)
  (object message))
```

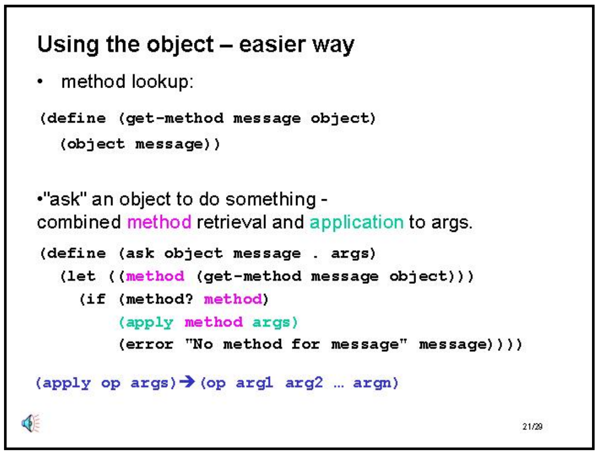
•"ask" an object to do something - combined method retrieval and application to args.

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

20/29

Slide 14.3.20
Now what does that mean? Remember that our methods could take different numbers of arguments, but we want one standard, clean interface. So in the definition of `ask` we use the "dot" notation to bind the variable `args` to a list of the values of any remaining arguments (other than the ones for `object` and `message`). To evaluate `method` applied to that **list** of arguments, we use a particular form, called `apply`.

Slide 14.3.21
Think of evaluating `apply` followed by an operator followed by a list of arguments as being equivalent to evaluating the expression `(operator argument1 argument2 ...)`. In other words, `apply` converts the operator and the list of arguments into exactly the form one needs for evaluation of an expression using the operator on the arguments.
So back to `ask`. It tries to get the method associated with a message, and if there is one, it executes the evaluation associated with using that method on the set of arguments, independent of how many there are. Otherwise, it complains that there is no method for this message for this object.



Using the object – easier way

- method lookup:

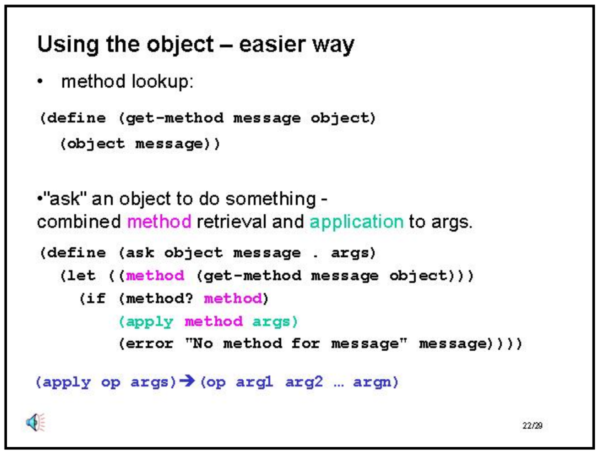
```
(define (get-method message object)
  (object message))
```

•"ask" an object to do something - combined method retrieval and application to args.

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

(apply op args)→(op arg1 arg2 … argn)

21/29



Using the object – easier way

- method lookup:

```
(define (get-method message object)
  (object message))
```

•"ask" an object to do something - combined method retrieval and application to args.

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

(apply op args)→(op arg1 arg2 … argn)

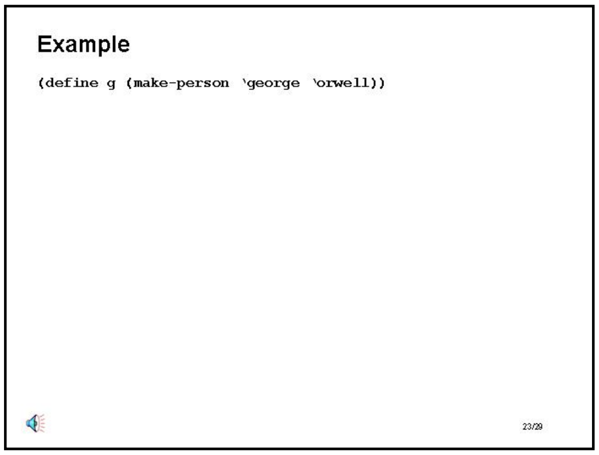
22/29

Slide 14.3.22
Step back for a second from the details of how we are implementing this system in Scheme from the higher level issues of what computational behavior we are building. So why do all of this?
In designing our OOPS system, we have chosen a convention in which all messages return a method. This makes it easier for the implementer, as she doesn't have to remember what kind of return convention one has for different messages. In all cases, a message returns a method. To do this, though, we also have to isolate the lookup of a method from the application of a method, and we have chosen to build a generic way of doing this. `Ask` becomes our standard interface to an instance and a method. Within this procedure we separate the idea of getting a method from the idea of using that method. But with this generic interface, we can ask any object to do any action.

Slide 14.3.23

Let's look at this in action. Let's again create a person, named George, that we bind to the variable `g` in the global environment.

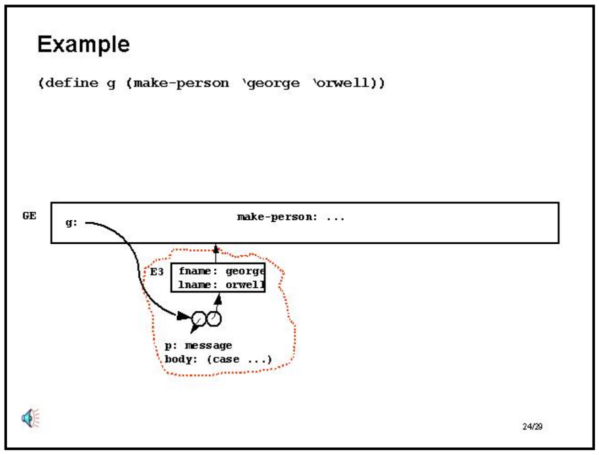
Let's see what the environment diagram for this looks like.



Example

```
(define g (make-person 'george 'orwell))
```

23/29



Example

```
(define g (make-person 'george 'orwell))
```

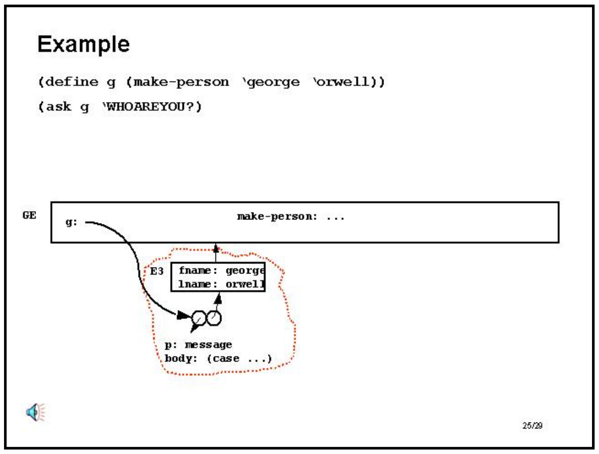
24/29

Slide 14.3.24

Just like our earlier examples, we see a form we would expect. The highlighted structure is a procedure that takes in messages, and which points to a local frame in which state information is stored. If you are not certain about how this is created, walk through the details of the application of `make-person`. In the global environment, `g` is bound to this object, this procedure with local state.

Slide 14.3.25

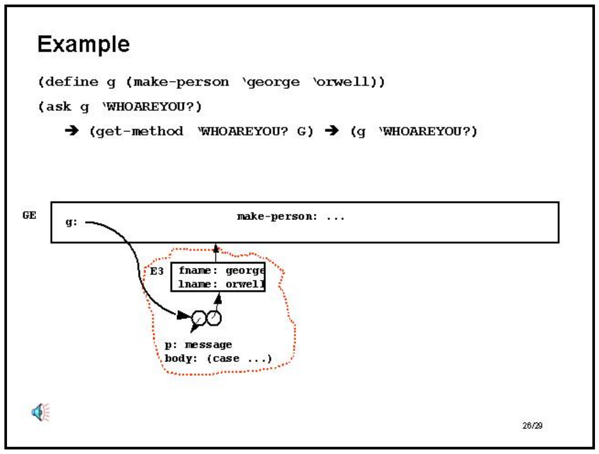
Now, let's ask `g` for its name.



Example

```
(define g (make-person 'george 'orwell))
(ask g 'WHOAREYOU?)
```

25/29



Example

```
(define g (make-person 'george 'orwell))
(ask g 'WHOAREYOU?)
  ➔ (get-method 'WHOAREYOU? G) ➔ (g 'WHOAREYOU?)
```

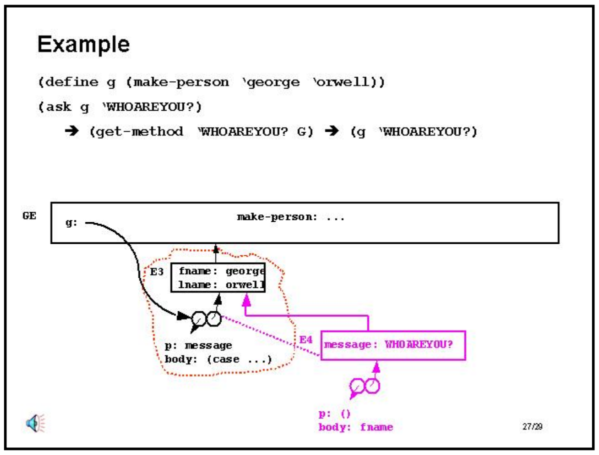
26/29

Slide 14.3.26

Recall that `ask` will first get the method associated with this message and object, by calling `get-method` on those arguments. That in turn reduces to evaluating the expression `(g 'WHOAREYOU?)`, and this in turn reduces to applying the procedure object `g` to the message `WHOAREYOU?`.

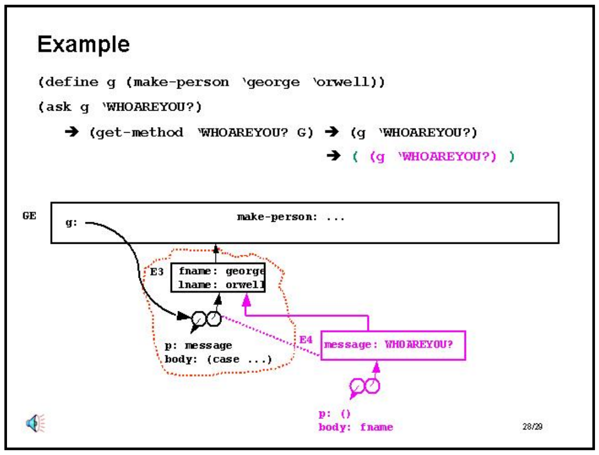
Slide 14.3.27

But for this implementation, we know what that does. `g` is a procedure and we are applying it a single argument, so we drop a frame, scoped by where the procedure's environment pointer points, and within this new environment we bind the parameter message to the symbol `WHOAREYOU?`. E4, this new environment, now becomes the environment in which we will evaluate the body of that procedure. That is just a big `case` expression, and eventually it returns a method, a method with no parameter and a body that is just `fname`. You can see this by looking back at the definition of `make-person`.



```
(define g (make-person 'george 'orwell))
(ask g 'WHOAREYOU?)
  ➔ (get-method 'WHOAREYOU? G) ➔ (g 'WHOAREYOU?)
```

Thus `ask` has sent a message to an object through `get-method`, which has created an internal frame that captures information about the local environment (i.e. it points to the same frame as the procedure) and relative to that we have obtained a new object, a procedure that represents the actual method.



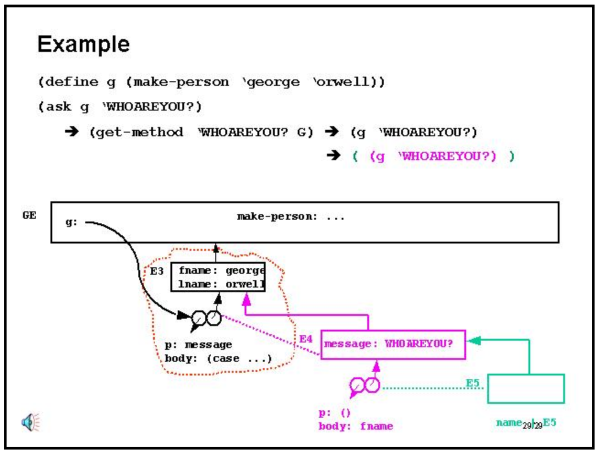
```
(define g (make-person 'george 'orwell))
(ask g 'WHOAREYOU?)
  ➔ (get-method 'WHOAREYOU? G) ➔ (g 'WHOAREYOU?)
                               ➔ ( (g 'WHOAREYOU?) )
```

Slide 14.3.28

Having gotten a method back,`ask` then **applies** it. And that is equivalent to taking the value returned by asking `g` its name, and evaluating it as a combination with no arguments. The actual details are slightly different but are equivalent to evaluating the expression shown.

Slide 14.3.29

And so an application of this method is simply a case of applying this procedure to no arguments. That again drops a frame, now scoped by the same frame as the procedure, namely E4. Relative to this new environment, E5, we evaluate the body of that procedure, which simply says to evaluate `fname`, i.e. lookup the binding for the symbol `fname` starting in E5. Thus, we trace up the environment chain through E4 to E3 where we find the binding. This then returns that value, the symbol `george`. Notice the two things going on here.`Ask` first gets a method, which creates for us a procedure with a scoping of local frames. Secondly, the application of that procedure representing a method causes a new environment, with appropriate scoping to be created so that evaluating the body of the method procedure has access to the local information of the instance.



```
(define g (make-person 'george 'orwell))
(ask g 'WHOAREYOU?)
  ➔ (get-method 'WHOAREYOU? G) ➔ (g 'WHOAREYOU?)
                               ➔ ( (g 'WHOAREYOU?) )
```