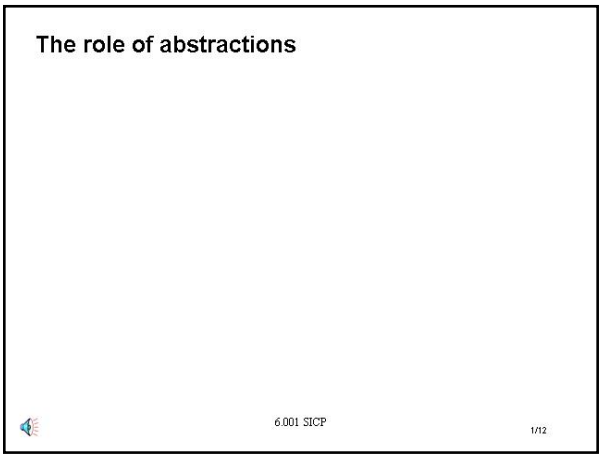
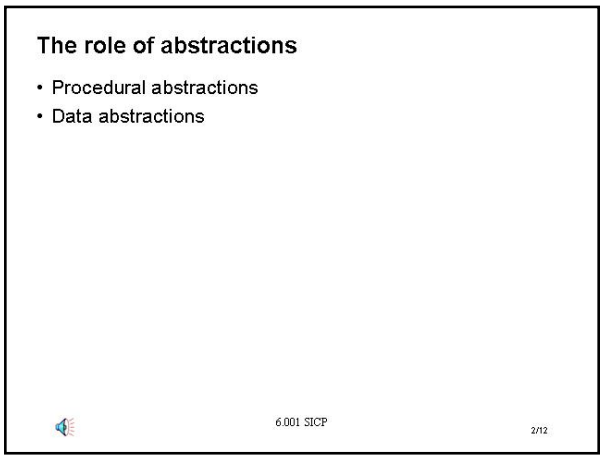
# 6.001 Notes: Section 13.1

**Slide 13.1.1**




In this lecture, we are going to look at a **very different** style of creating large systems, a style called **object oriented programming**. This style focuses on breaking systems up in a different manner than those we have seen before. To set the stage for this, we are first going to return to the notion of abstractions, and use that idea to see how we can capture objects with some internal state that reflects the status of those objects. We are going to be led from there to a style of programming called **message-passing** in which we treat systems as if they consist of large collections of objects that communicate with one another to cause computation to take place.




**Slide 13.1.2**

Let's start by going back and thinking about the tools we have developed for thinking about computation. Two of the key tools we have developed dealt with abstractions.

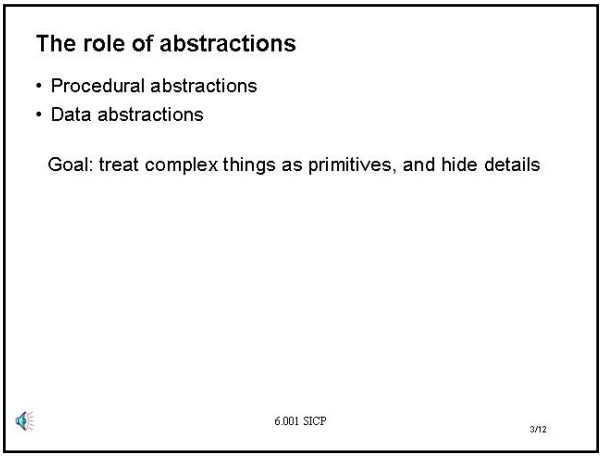
We have seen **procedural abstractions**. Here the idea is to capture a common pattern of processing into a procedure, then isolate the details of the computation from the use of the computation, by simply naming the procedure and using that name with appropriate conditions on the procedure's input. We saw that this style of approach is particularly useful when dealing with problems that are easily addressed in a functional programming approach, that is, where we can treat the procedures as generalized mathematical functions, meaning that their output for a given input will be the same whenever we evaluate it.

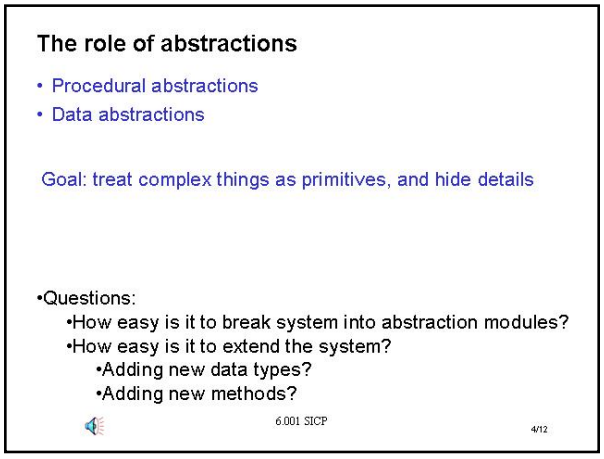
We have also seen **data abstractions**. Here the idea is to modularize our system by creating data structures that capture key parts of the information we need to handle. The goal is to hide the details of the representation and storage of the data behind standard interfaces, primarily our constructors and selectors. This means the user can then manipulate data objects without having to worry about details of how they are maintained.

As you might expect, often the data abstractions and the procedural abstractions work hand-in-hand, with the procedures used to manipulate the data using the data abstraction interfaces, and with the structure of the procedures tending to mirror the actual structure of the data.

**Slide 13.1.3**

The goal in each case is actually the same: we want to hide details of the abstraction so that we can treat complex things as if they are primitive units. In the case of procedural abstractions, we want to hide the details of the computation, and treat the procedure as a primitive computational unit. In the case of data abstractions, we want to hide the details of how components are glued together, and treat each unit as an abstract collection of parts.



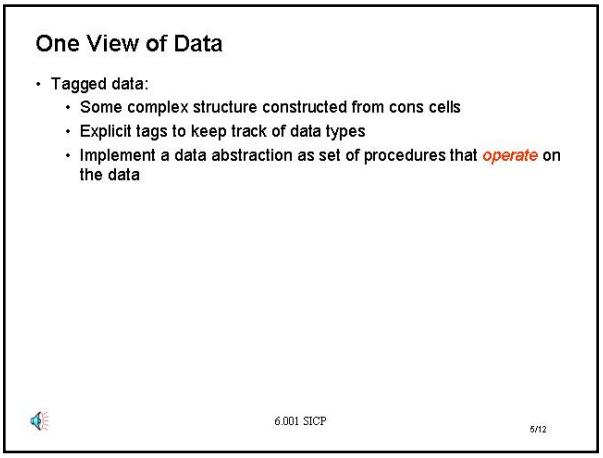



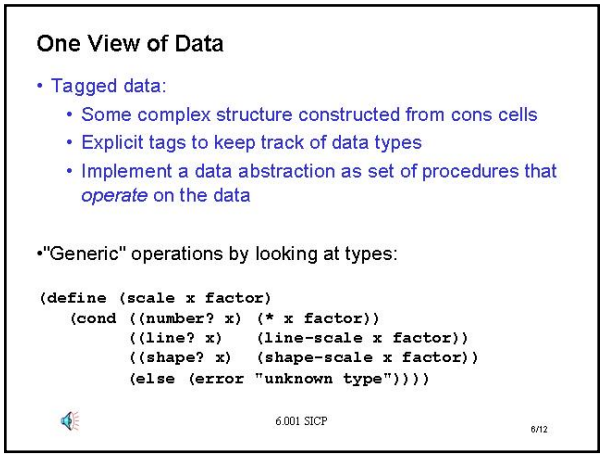

**Slide 13.1.4**

Given that we want to use abstractions as a tool in controlling complexity in large systems, there are several questions that come up when thinking about how to use abstractions. The first is: what is the best way to break a new problem area up into a set of modules? Both data modules and procedure modules? As we have already seen in earlier lectures, some problems break up in multiple ways, and breaking them up in different ways makes some processes easier and others harder. So a key question is: How do I use the idea of abstraction to break systems into modules and what's the best way to do this?
The second question deals with how easy it is to extend the system. If I want to add new data types to my system, is that easy? If I want to add new methods to my system, new ways of manipulating data types, is that easy? We have seen several examples of this already, we are now going to return to these questions in order to lead to a very different way of breaking systems up into convenient sized chunks.

**Slide 13.1.5**

Let's start by going back to data objects and data abstractions. Here is the traditional way of looking at data, at least as we have done things so far.
First, we build some complex data structure out of primitives, for example, `cons` cells or pairs. Second, we use tags to identify the type of structure being represented. This tells us how to interpret the different slots in the list structure. For example, is the `car` of the list structure the name of a person or his batting average or his GPA?
Then, the data abstraction is actually built by creating a set of procedures that operate on the data. These are procedures that take in instances of the data, use selectors to get out the pieces, do some manipulation to create new pieces, and then use the constructor to re-glue the abstraction back together. This led to the concept of data-directed programming, which we saw earlier. We use the tag to determine the right set of procedures to apply. And this allows the user to program in a generic fashion. They can focus on **what** they want to do, but have the code direct the data to the right place for the actual work.



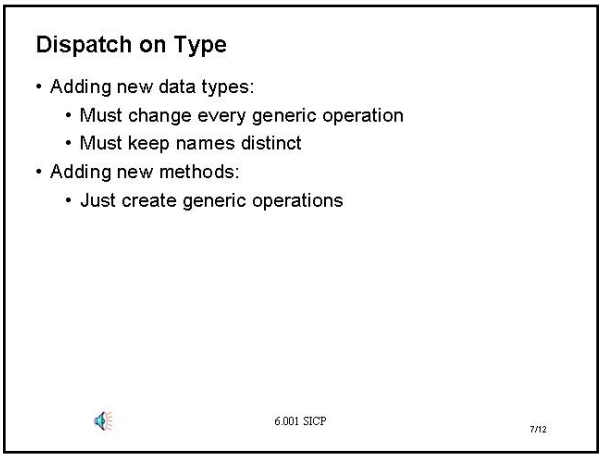



**Slide 13.1.6**

Here is a simple example to illustrate this point. Suppose I have a set of different geometric objects, things like numbers, lines, shapes, and I want to write a procedure, or an operation, that will scale each of those objects by some amount. Then a generic operation, under the data-directed approach would look like this procedure shown here. Given an object and my desired scale factor, I use the type of the object to dispatch: if it is a number, I just multiply; if it is a line, I ship it to the procedure that will scale a line, and so on.

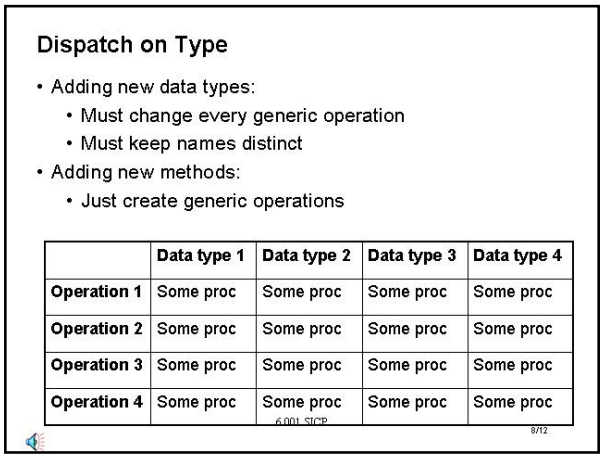
The point of this example is that I think about things in terms of the kinds of objects I have and procedures for manipulating each distinct object type. I use the tag or the type of the object to tell me which procedure to send the object to.

**Slide 13.1.7**




So now let's go back to our questions. How easy is it to extend such a system, a system where we are breaking things up into tagged data, and using data directed programming? First, if we add a new data type to our system, what do we have to do? Well we can see from our example that we will have to all the procedures like `scale`, to add a new clause to each `cond`, dispatching on that new type of object. As a consequence, if there are many such procedures, we have a lot of changes to make, both a great deal of code to write, and more importantly making sure that we change all the relevant procedures.

If we add a new operation or method, what do we need to do? This is easier, as we just need to develop a subprocedure for each type of object to which the method will apply. Thus in this style of programming, adding a new data type is painful, while adding a new method is reasonable. As a consequence, this approach to modularizing systems works best when there are only a few data abstractions or when the changes are mostly new methods or operations, or when the different kinds of data structures in the system are mostly independent of one another. In those cases, this style of approach works well. But not everything fits these cases. What should we do in those cases?

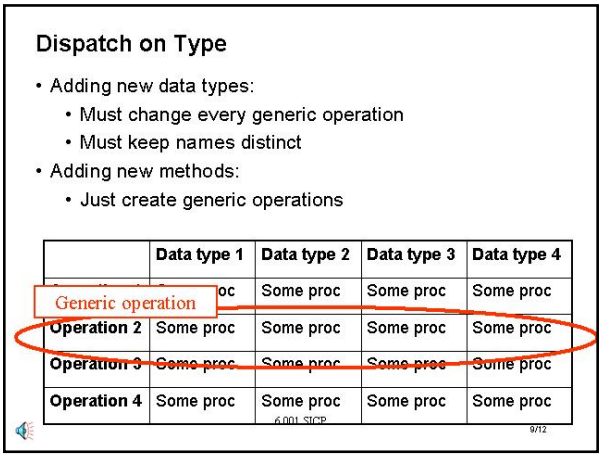


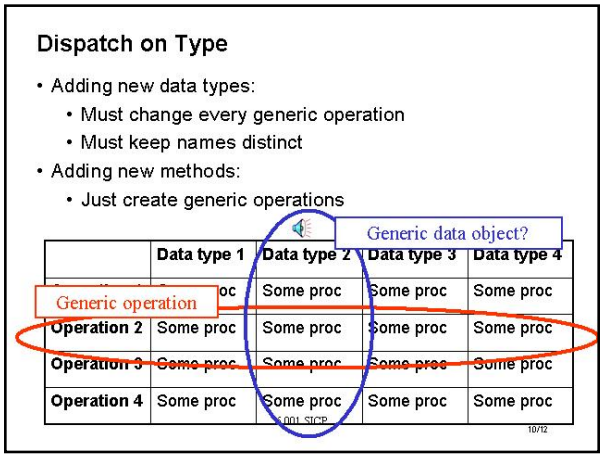

**Slide 13.1.8**

So let's step back from this organization for a second. One way to think about structuring a large system is to realize that we are likely to have a large number of different data objects (or instances of data abstractions), and a large number of operations we want to perform on those objects. Conceptually, this means we have a big table, where we can use a different row for each operation we want to perform, and a different column for each kind of data abstraction we have. Then at each element in this table, we can conceptualize having a specific procedure, intended to perform the particular operation (e.g. scaling) on the particular kind of data object (e.g. a number).

**Slide 13.1.9**

One way of actually building such a system is to focus on the rows of the table, that is the operations. Indeed, our use of tagged data was based around this viewpoint, in which we created **generic operations** that handle the same operation for different data objects, and used the tag on the data object to **dispatch** to the appropriate version of the procedure to handle that kind of data.



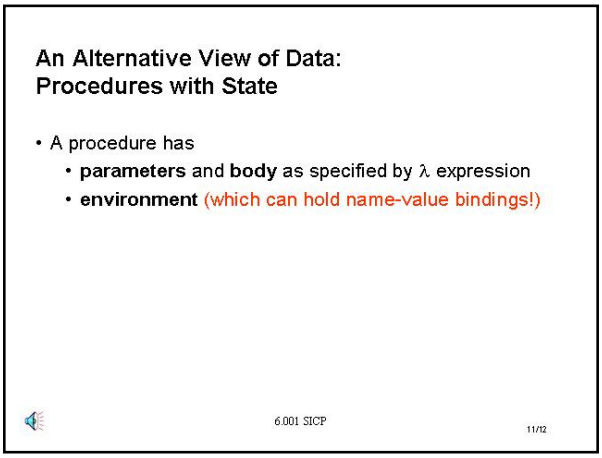

**Slide 13.1.10**

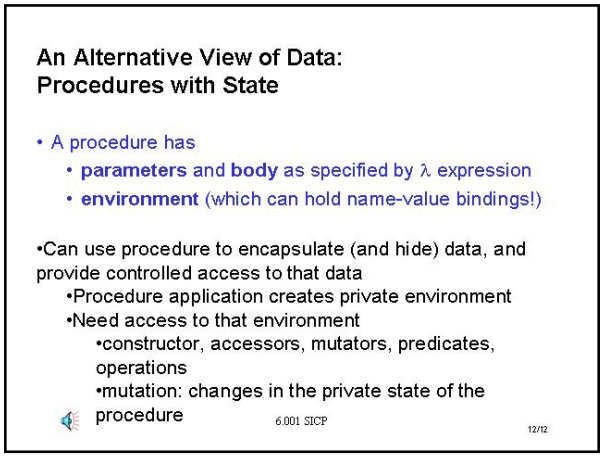
But given this table, there is an alternative possible organization, which is around the columns of the table. This would focus on creating a **generic data object** that would know how to handle different operations on that kind of data structure.

**Slide 13.1.11**

Let's step back and **rethink** data. This sounds like an odd thing to do but let's think about data in a very different way. Rather than thinking of data abstractions as some slots into which we can put things, let's instead consider data to be a procedure with some internal state.

This sounds strange! But, what is a procedure? It really has two parts: it has a set of parameters and a body which define the pattern of computation to perform as a function of the objects passed in; and as we saw in the environment model, it has an associated environment which can hold name-value bindings, that is, pairings of names and values.

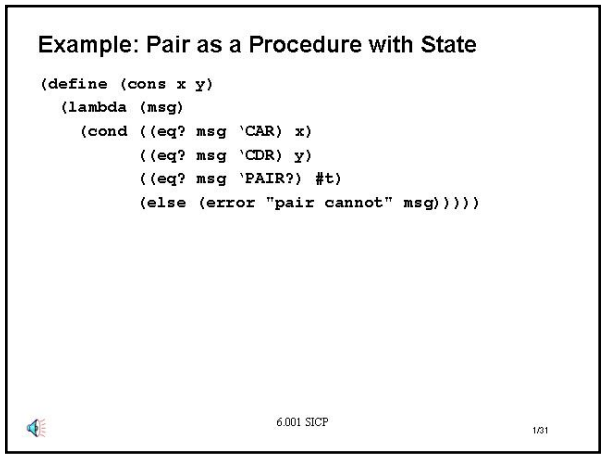


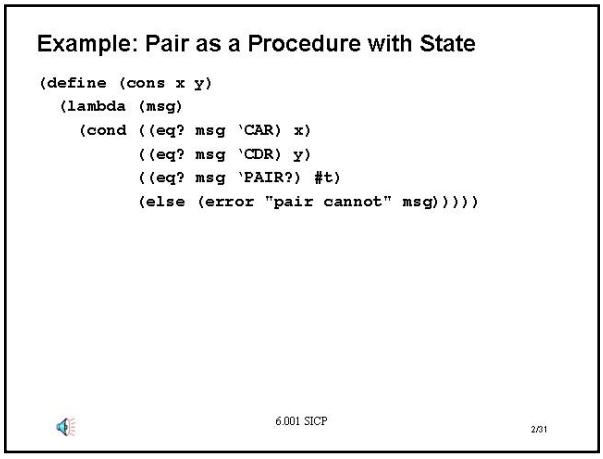



**Slide 13.1.12**
So what, you say! Well, we can use this idea to capture information about a data structure. In particular, we can use a **procedure** to represent data objects with state. What would that mean? It would say that we could use the local environment of the procedure plus its parameters to hold the values of the datum, and we could create local procedures within the data procedure to manipulate these values, to change the state of the object.
This means that the only access to the values of the data object will be through the procedure representing the data. This would nicely encapsulate the data structure inside this procedure.
This probably still sounds odd so let's look at a specific example.

# 6.001 Notes: Section 13.2

**Slide 13.2.1**
To illustrate this idea of using a procedure to represent a data structure, an object with state, let's look at the following, rather odd, example. Here is a very different way of implementing a `cons` cell or a **pair**. Let me stress that this is **not** the way that Scheme normally represents pairs. Of course, the idea of data abstraction is that the actual implementation of a data structure should be irrelevant to the user. This example is used to drive home a conceptual point.
Here, we have implemented a pair as a **procedure**! Thus our fundamental data structure is now a procedure rather than some storage in memory slots.

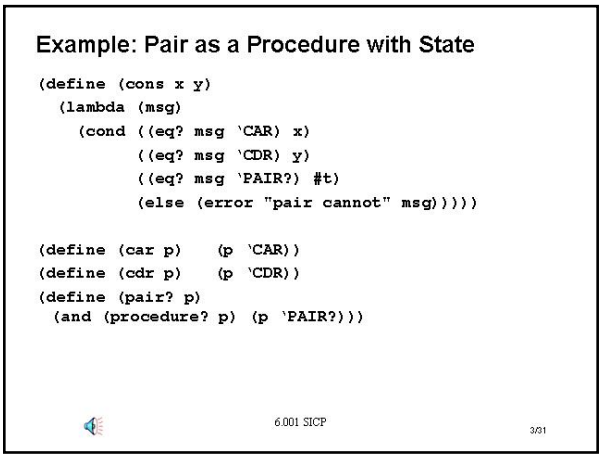



**Slide 13.2.2**
Look at this carefully. First, note that `cons`, as defined here, involves **two** lambdas. Remember that there is a hidden `lambda` inside the syntactic sugar of this definition. This means that there is a second `lambda` as the body of the `cons` and thus when we evaluate `(cons x y)` using this particular implementation, we get back as a value, a procedure of one parameter, `msg`.
So what does this say? It says that when we use `cons` with this implementation our representation for our fundamental way of gluing things together is now a procedure of one argument. So what would that `cons` thing do? Since it is a procedure, if we send it a value, or if we apply the procedure to a single argument, note what it does. It uses the value of the argument, in this case a particular symbol, to decide

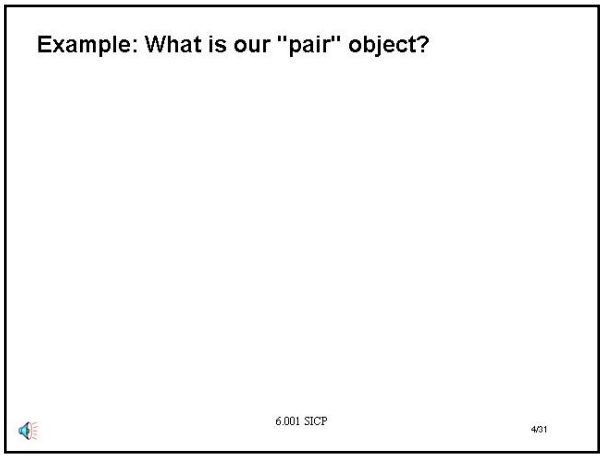
what value to return.
We call this style of programming, **message passing**, because the procedure accepts a message as input, and then does something based on the value of that message.

**Slide 13.2.3**




This looks a bit weird! Our constructor for gluing things together gives us a procedure as the actual object. Should we care?
Of course we know that we shouldn't care. To complete the abstraction for a pair, we simply need to create `car` and `cdr` to fulfill the contract of the abstraction of a pair.
Each of those is itself a procedure that takes as input a pair, which we know is a procedure, and then applies that procedure to a single argument, which in this case is just a symbolic message. Ideally, that message should get back for us the value we need to satisfy the contract. If we look at this definition for `car`, we see it takes as input one of these new pairs, and then applies that pair (a procedure) to the symbol `car`, `which` in principle should return for us the value we used when we created the pair.

Note the other procedure we built here. Our predicate for testing whether something is a pair now relies on the pair identifying itself. This is the version of our tag. Before we attached a tag as a symbol on a data structure. Here, our tags are part of the procedure.

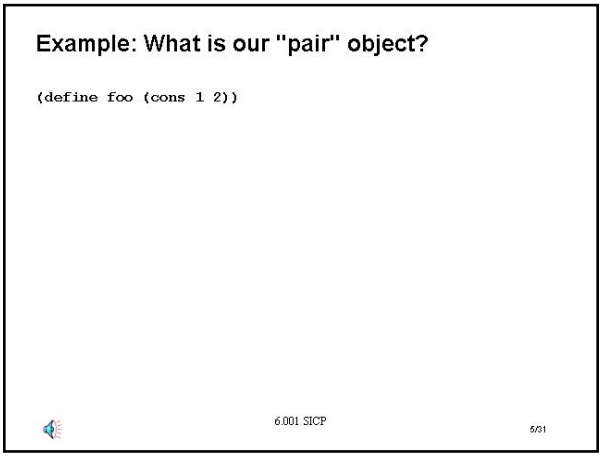


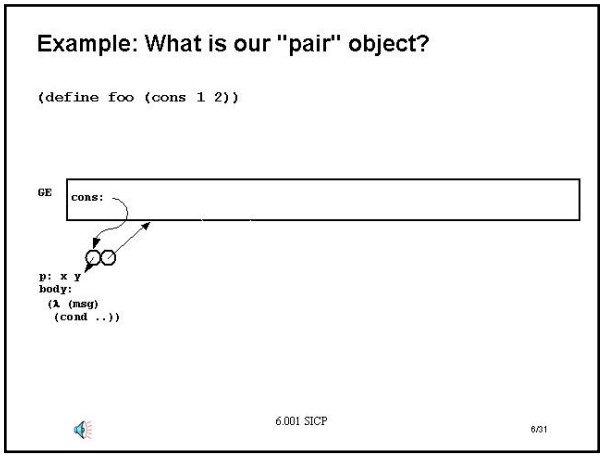

**Slide 13.2.4**

To check it out, let's take this strange implementation of pairs and verify that this implementation satisfies the contract for a pair.

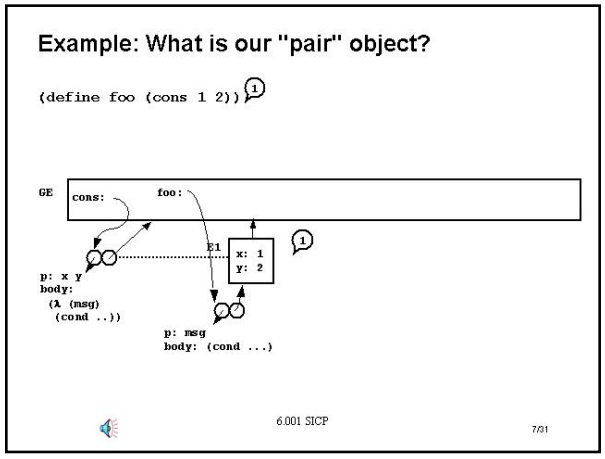
**Slide 13.2.5**

To test this, lets cons together the numbers 1 and 2, and give the resulting pair the name `foo`.

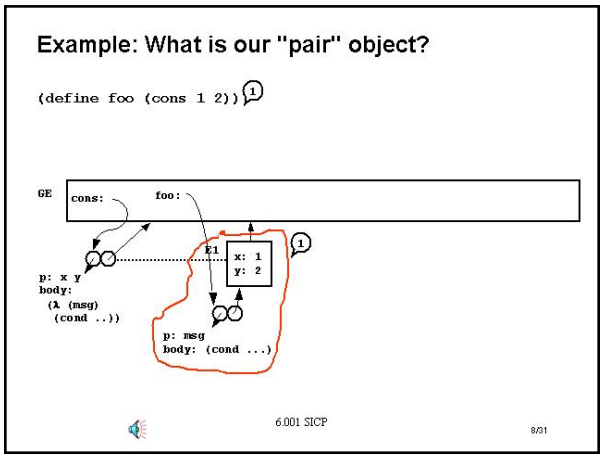





**Slide 13.2.6**
So here is an environment diagram that would represent the state of the world, before we do this definition. In the global environment, we would have a binding for `cons` as a procedure, based on the previous slide.




**Slide 13.2.7**
What happens when we evaluate this expression? Since `cons` is just a procedure, evaluating `(cons 1 2)` says to apply the procedure associated with `cons` to the arguments 1 and 2. Thus, we drop a frame, scope it by the environment pointer of the procedure, bind the formal parameters (`x` and `y`) of the procedure to the values of the arguments, and relative to that new frame, and evaluate the body of the procedure. That body is itself a `lambda`! So it makes a new procedure object, whose environment pointer points to the frame E1 because that is where the `lambda` was evaluated. Then, the procedure object is returned as the value of the `cons`. Finally, the `define` binds `foo` in the global environment to this returned value, this procedure object.

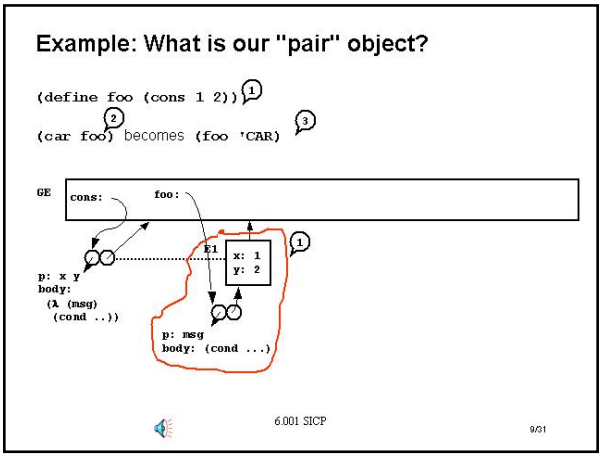


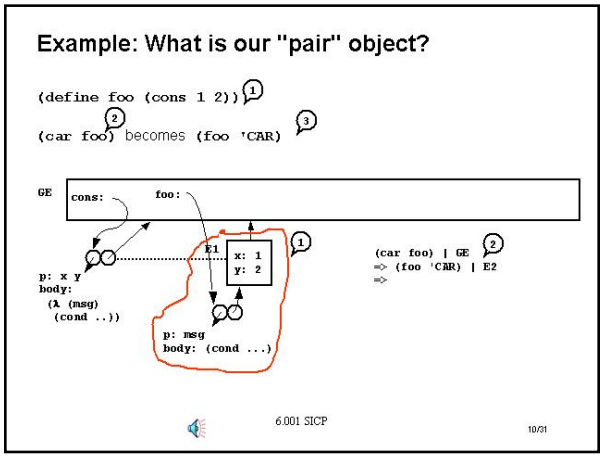

**Slide 13.2.8**
Notice what this does. It gives us an object in this environment, where by object I mean the thing enclosed in red, which is a procedure that has a local frame with some bindings or values within it. Thus, `x` being bound to 1, and `y` being bound to 2 constitutes local state information. That frame is scoped by the global environment, and the procedure that points to all of this is referred to by a name in the global environment. Thus, from the perspective of a user interacting at the global environment, `foo` refers to a structure that has within it information about what is the first part of the object (1) and what is the second part of the object (2). It should also have information about how to extract those values from the structure.
So this pattern: of a procedure that accepts messages, has access to a local frame with state and methods to extract that local state; is a very common pattern that we are going to use a lot.

**Slide 13.2.9**

Now all we have to do is check that the contract holds for this data abstraction. In doing so, we will see how this structure of a procedure with access to local state captures exactly the behavior we want.

To check this, lets evaluate `(car foo)`. We know that this should get converted into `(foo 'car)`, so how does this happen?



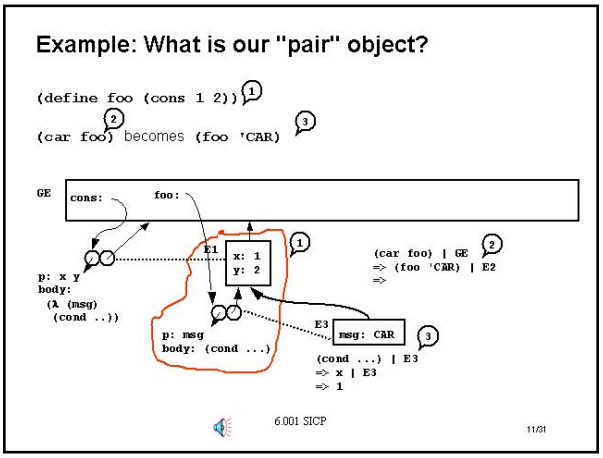



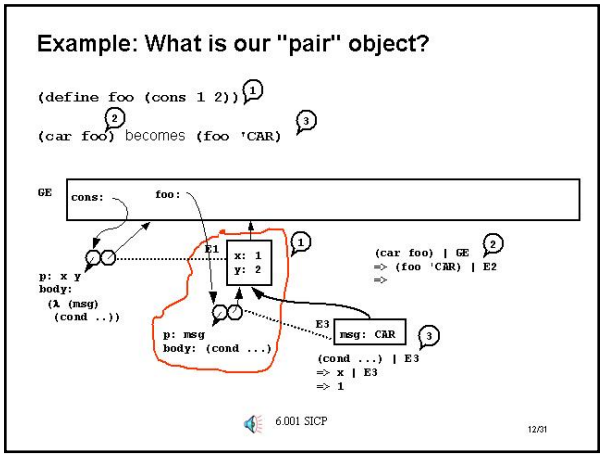

**Slide 13.2.10**

Evaluating `(car foo)` in the global environment simply applies the procedure that is the value associated with `car` to the value of `foo` which is the procedure object shown. Now the definition of `car` shows that this reduces to evaluating the body of `car` namely `(foo 'car)` with respect to some new environment.

**Slide 13.2.11**

... and what does that do? It says to apply the value associated with `foo`, which is a procedure, so the standard environment model says to drop a frame, and scope it by the environment pointer of `foo`. This is important as E3 now points to E1. Inside E3 we bind the parameter `msg` to the argument `car`. Relative to this frame we evaluate the body of the procedure represented by `foo`. But that is just a `cond` clause that looks at the value of `msg` and compares it to a set of symbols. In this case, the `cond` says to return the value of `x` with respect to this frame, which is just `1`. This is exactly what I wanted, as it shows that my contract is satisfied.



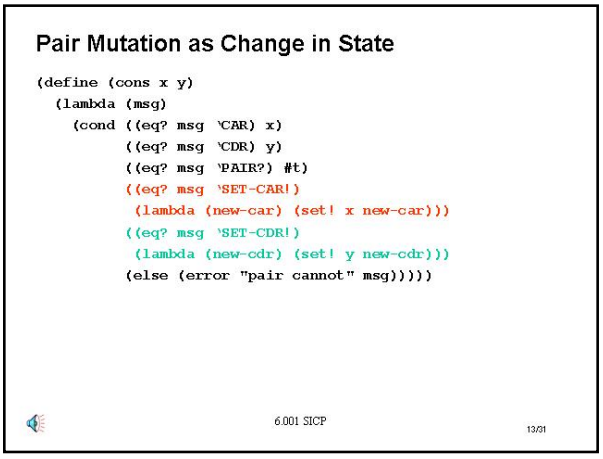


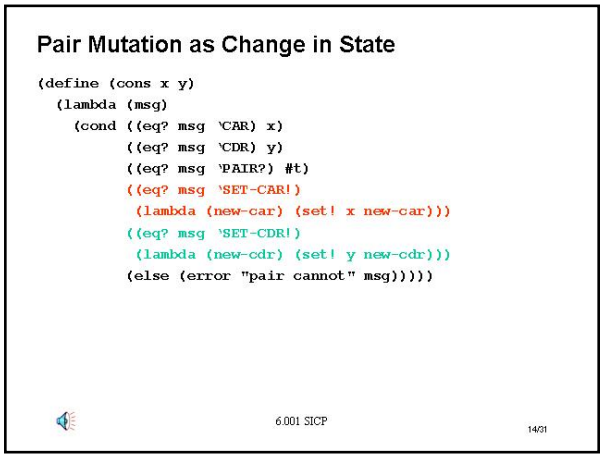

**Slide 13.2.12**

So what does all this say? Aside from showing that our contract is fulfilled, that what we glue together using this version of `cons` we can get back apart using `car` or `cdr`, we have also seen this common pattern that we can create a data object represented as a procedure. The procedure has some local state captured in a frame that is accessible only by that procedure and it has the ability to accept messages and based on those messages return information from the local state. So let's see how to build on that idea.

**Slide 13.2.13**

In the case we just considered, our procedures for data structures could return values as a function of input messages. If we are going to use this idea of message-passing procedures to represent information, we also need to have ways of changing the value of the state captured by those procedures. In our pair example, here is how we would do this.



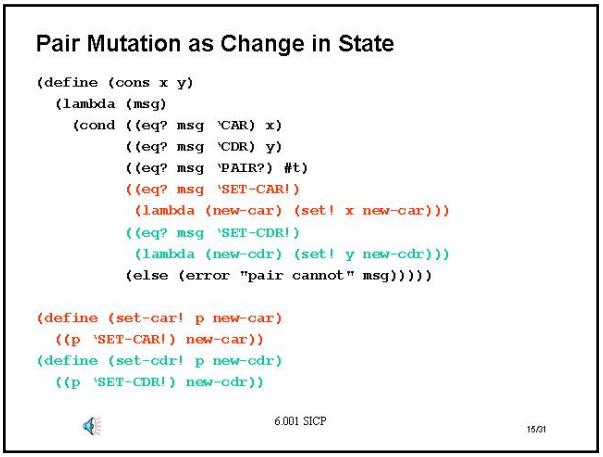

**Slide 13.2.14**

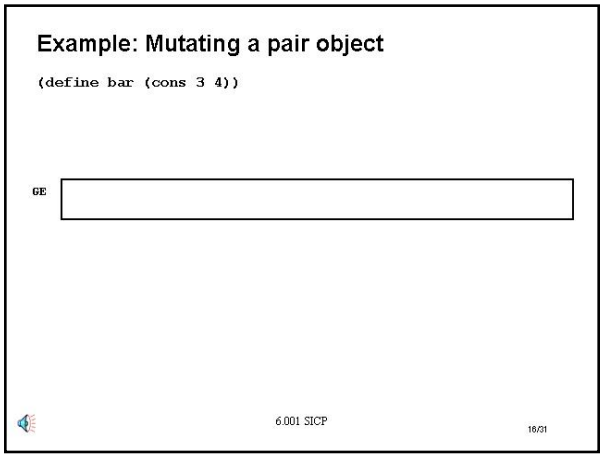
Let's add two more messages, or two more ways of dealing with messages, to our constructor, `cons`: one for dealing with mutating the `car` and one for dealing with mutating the `cdr`. Notice that in this case we need something different. If the `cons` pair (i.e. one of these procedures) gets the message `set-car!` we are going to return a **procedure** that will take a new value for the `car` and change the old value to this new value.

This is a different behavior from before. Now a message gets us back a procedure rather than a number.

**Slide 13.2.15**

As a consequence, the procedure `set-car!` must have a new form. As before, it will take a pair and a new value as arguments, but now it sends the pair (that procedure) the message `set-car!`, which gives us the procedure needed to change values, and we then apply **that** procedure to the new value. You can see that the definition accomplishes exactly this.



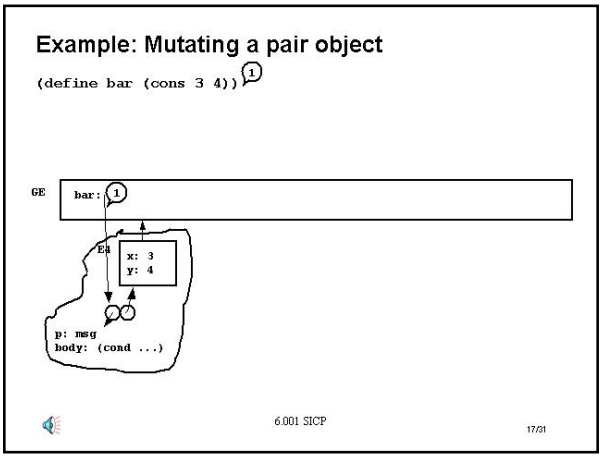



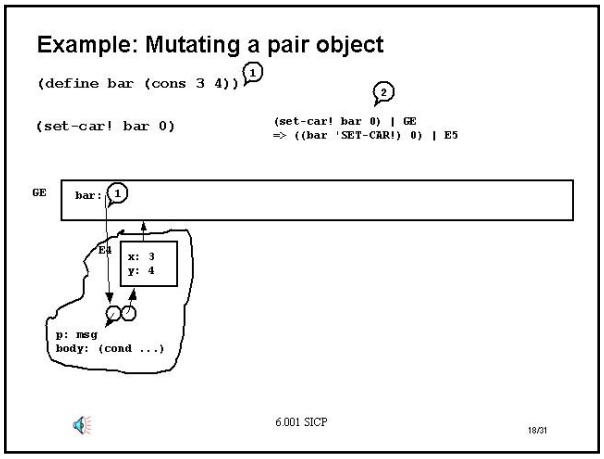

**Slide 13.2.16**

So let's trace this through. Here is a definition for `bar` to be the `cons` of 3 and 4, and here is the global environment in which we are going to do this.

**Slide 13.2.17**

When we evaluate this expression we simply get a structure similar to what we saw before, a binding of `bar` to a procedure with some local state. Thus, we have `bar` as a message-passing object.



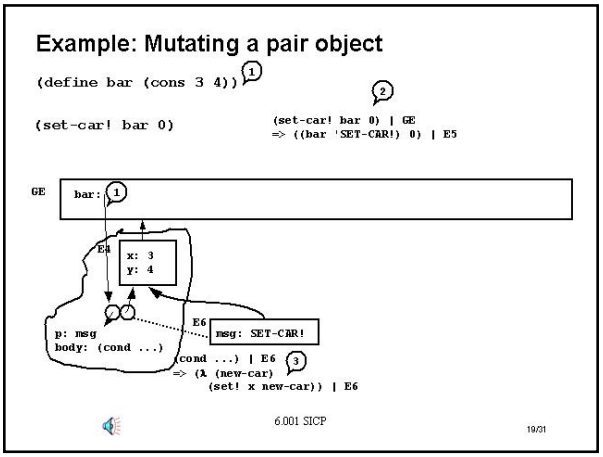



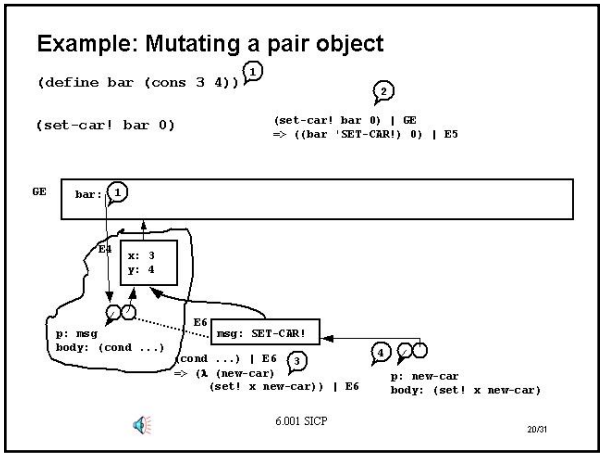

**Slide 13.2.18**

So now let's mutate this object. Let's change the `car` part of this object to be 0. Then evaluating `(set-car! bar 0)` reduces to evaluating `((bar 'set-car!) 0)` in some other frame. Now how does evaluating this expression effect the right mutation?

**Slide 13.2.19**

First, we need to get the values of the subexpressions with respect to this frame. Well `bar` is bound to a procedure, so we can apply it to the symbol `set-car!`. This drops a frame, scoped by E4 because `bar`'s procedure is also scoped there. Within that frame we bind `msg` to the symbol `set-car!` and relative to that frame we evaluate the body of the procedure `bar`. This will return an expression `(lambda (new-car) (set! x new-car))` to be evaluated with respect to this frame E6.



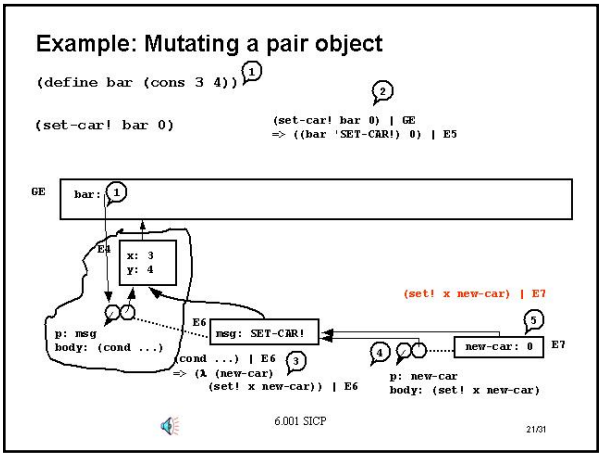

**Slide 13.2.20**

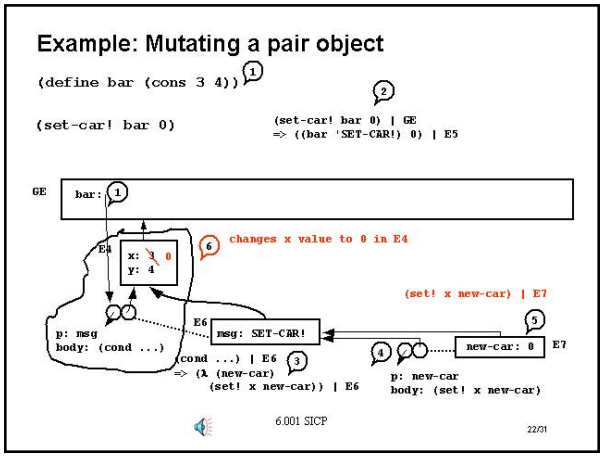
Now here comes the critical point. Remember that we are evaluating the body of `bar` with respect to E6, which reduced to evaluating `(lambda (new-car) (set! x new-car))` with respect to this frame E6.

This, of course, creates a procedure object, whose environment pointer is scoped by E6 (and this is the crucial point!). Note that this newly created procedure object will have access to E6 and by chaining to E4. This procedure object is the value returned by evaluating `(bar 'set-car!)`.

**Slide 13.2.21**

... and this is exactly what I want. I now apply this procedure to the value 0, which is the last part of the original evaluation. By environment model, I just drop a frame, scoped by E6, binding new-car to the value 0. Thus E7 is scoped by E6, which in turn is scoped by E4. Relative to E7, I now evaluate the body of this procedure that I just created, and that says `(set! x new-car)` with respect to E7.



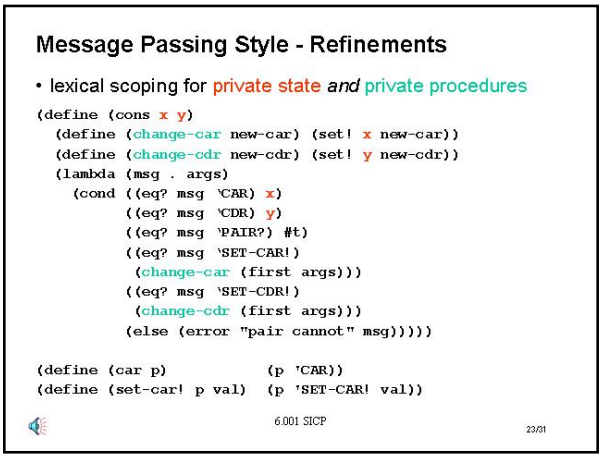




**Slide 13.2.22**

So now we evaluate that `set!` expression with respect to E7. First we find the binding for `x`, which we get by tracing from E7 through E6 to E4. Then we find the binding for `new-car` which we find in E7, and then change the binding for `x` in E4 to this new value.

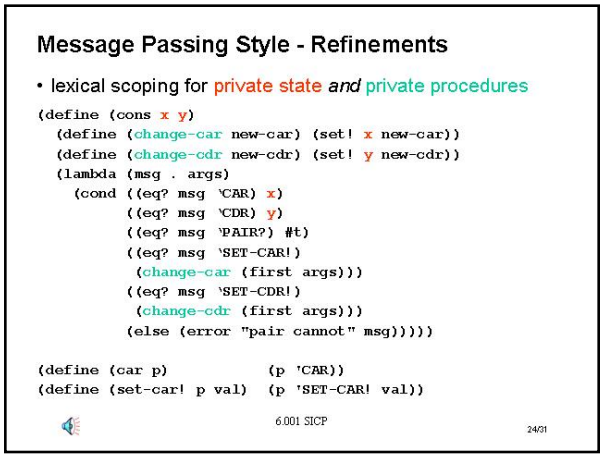
Notice how we have now mutated a value in the local state associated with `bar`. Thus our procedures can not only capture local state information, they can also supply procedures for changing local state.

**Slide 13.2.23**




This is certainly different from our earlier data abstractions. Now we have data objects that are actually procedures. A cons pair is now a procedure, and `car` or `cdr` is something that operates on a procedure. But as we have seen, the `cons, car, cdr` we just built satisfy the data abstraction contract, and therefore behave as expected. The key new thing we have is a procedure that represents data, and takes messages as input and returns either data values or procedures for changing data values. This is a very handy idea, so let's generalize it.

Let's create private state variables (as we did earlier) but also private procedures that will belong to each instance of the data abstraction.




**Slide 13.2.24**

Notice the difference in this version. Now I create internal procedures (`change-car, change-cdr`) but inside of my actual object, I not only create those procedures, I **use** them inside the object. This has a nice effect in that when I execute some operation on an object, I don't have to remember what type of value is returned by the object (e.g. number versus procedure). In all cases, the use of the data object is identical. Thus now our selectors and mutators perform in a uniform manner.

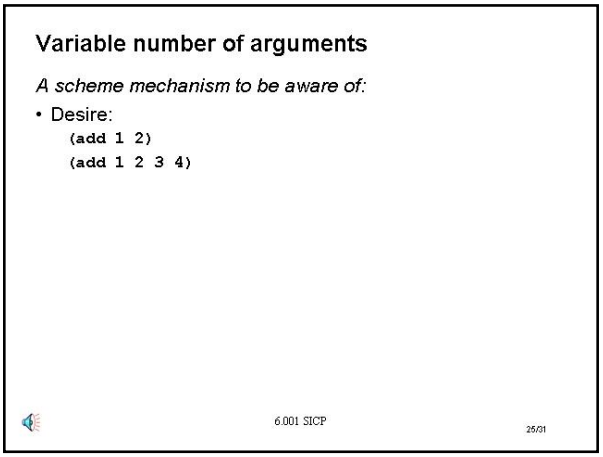
Before, we had to remember whether the object returned a value or a procedure, in order to complete our manipulation. Here, the selectors and mutators just send a message to the object and within the implementation of the object, we take care of the necessary work to either apply an internal procedure or to simply return a value.

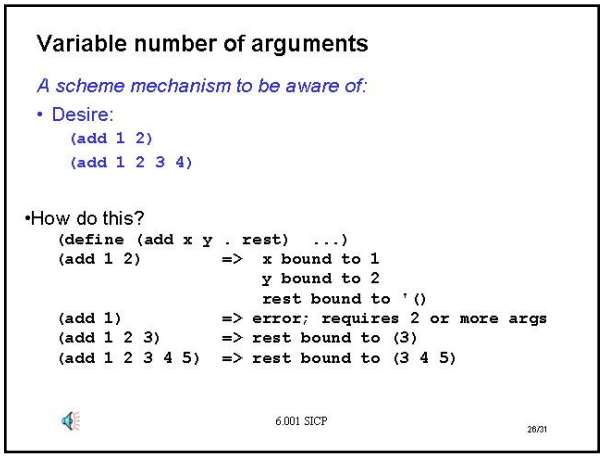
Notice that by defining the internal procedures within the context of the `cons` we will create procedures that are scoped within the created by calling the `cons`. Thus, these procedures will belong only to this instance of the data object.

By making a uniform interface for mutators and selectors we have introduced one other thing into our system. In particular we now need our selectors and mutators to deal with different numbers of arguments, and yet we would like our data abstraction to be a single procedure. So we need a way of letting a `lambda` specify that it wants to take an arbitrary number of arguments. That is the "funny" notation you see of `(lambda (msg . args) body)`, which we deal with on the next slide.

**Slide 13.2.25**

Up until now, every procedure you have written has required that you specify names of all the input parameters, and as you have seen, if you call the procedure with the wrong number of arguments it causes an error. We would like a mechanism that lets a procedure take an arbitrary number of arguments, such as you have already seen with built-in procedures like `+`.



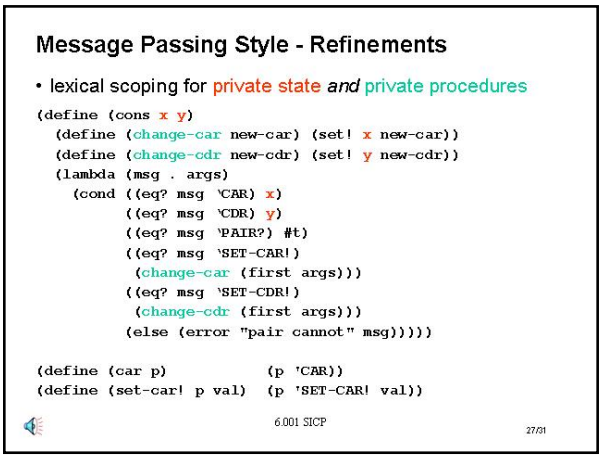



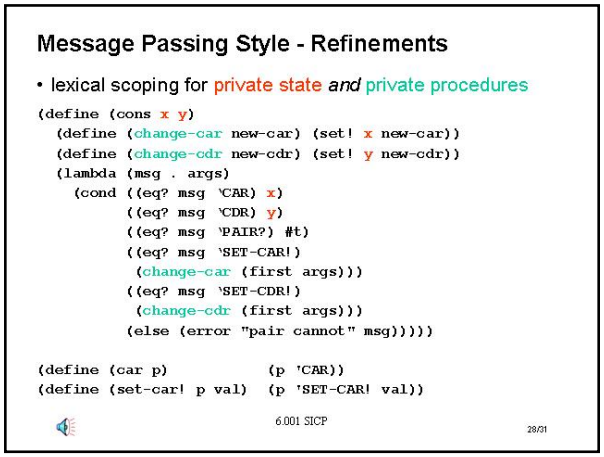

**Slide 13.2.26**

So Scheme provides a way of doing this. To see this, let's define `(add x y . rest)` to be a procedure that is going to add a bunch of things. Here, the syntax is an argument `x` an argument `y` and then a dot (`.`) and then the argument `rest`. And the behavior is as follows: If we apply `add` to a set of arguments, the value of the first argument will be bound to the variable `x`, the value of the second argument will be bound to the variable `y` and the values of the any other arguments will be bound, as a **list** to the variable `rest`. Thus if we evaluate `(add 1 2)` then `x` is bound to 1, `y` is bound to 2, and `rest` is bound to an empty list. If we try to evaluate `(add 1)` we will get an error, because in this format, the first two arguments are required, i.e. we must have something for both `x` and `y`. But if we evaluate `(add 1 2 3)`, then `x` will be bound to 1, `y` will be bound to 2, and `rest` will be bound to the list `(3)`. And if we evaluate `(add 1 2 3 4 5)` in this case `rest` will be bound to the list `(3 4 5)`. Thus in this notation, all of the parameters prior to the dot must have a value passed in, the parameter after the dot will be bound to the list of the values of all the remaining arguments.

**Slide 13.2.27**

If we come back to our example, we see that if we just take the `car` of a pair, `msg` will be bound to the symbol `car`, and `rest` will be bound to the empty list. On the other hand, if we want to mutate the car of a pair, then `msg` will be bound to the symbol `set-car!`, and `args` will be bound to the list of one value, the new value to be used. Within the procedure that defines the `cons`, notice what happens. If we are going to execute a `set-car!`, we apply the internal procedure `change-car` to the first value in the list `args`. In other words, it will extract the right value to use, and cause the appropriate change, it will mutate the binding of `x` to be that new value. And thus we can apply these internal procedures to the appropriate values, while preserving a uniform external interface.





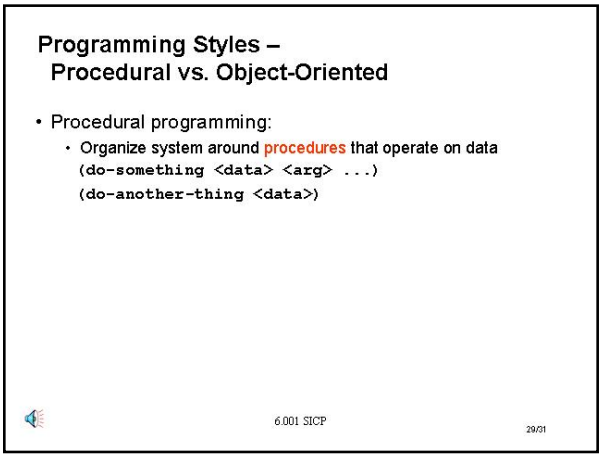

**Slide 13.2.28**

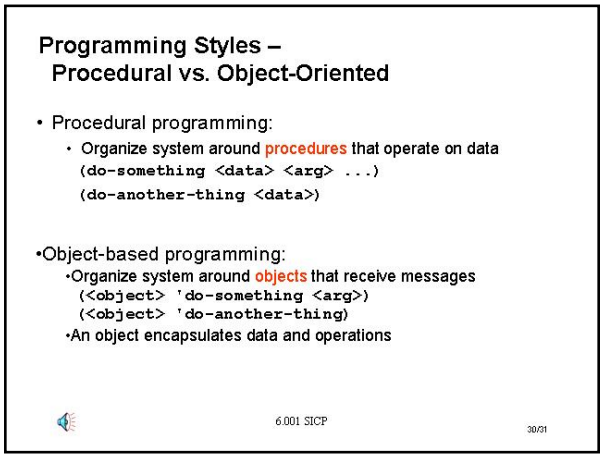
Since we have been throwing a lot of details at you, let's step back for a moment. What we have now seen is a method, using a particular example to illustrate, for creating a procedure that captures local state. This procedure takes in a message, and based on that message and possibly some other arguments, either returns values based on the local state or it causes changes in that local state.

This new method, this idea of a message passing procedure, let's us capture information about a data structure inside a procedure itself.

**Slide 13.2.29**

So what does this say? We have introduced the basic idea of a new style for approaching computational systems. Our traditional style is procedural programming: we organize the system around the procedures that operate on the data. The key is that we isolate the data in standard list-like structures, with tags, then focus on thinking about what methods or procedures we want to use to manipulate the values within those structures.



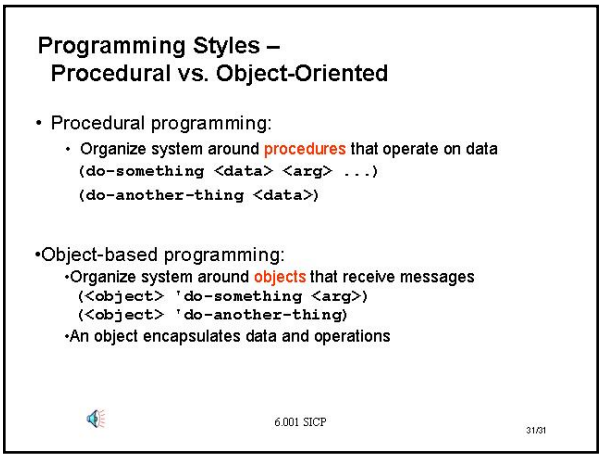

**Slide 13.2.30**

Here we have shown the basis for a new approach, which is oriented around the data objects themselves. Note what we did with our example of a pair. We focused on capturing the information within a structure, where the operations to manipulate the data were associated directly with that structure. More importantly, the basic conceptual unit was the data object itself. What messages should an object handle? What operations should an object support? How should we capture those methods internally within the object?

**Slide 13.2.31**

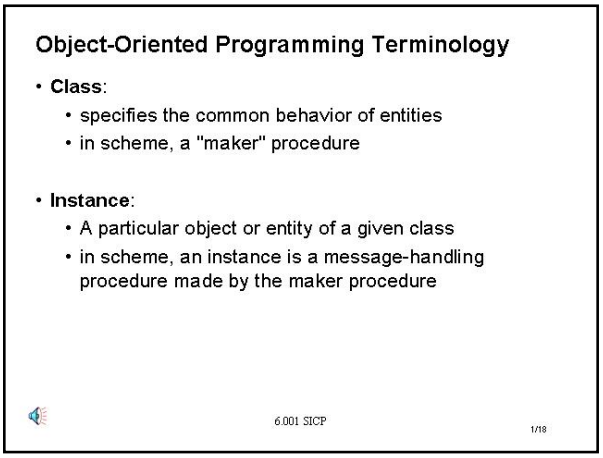
So which approach is better? It depends on the problem domain! Procedural methods are very good when we are dealing with things like numerical operations or when we are dealing with systems with very small numbers of data structures.

On the other hand, object oriented systems are very good for things like simulations or for systems with large numbers of objects, where the objects are characterized by a small amount of state information and the computation basically involves interaction between the objects, causing that state to change.



# 6.001 Notes: Section 13.3

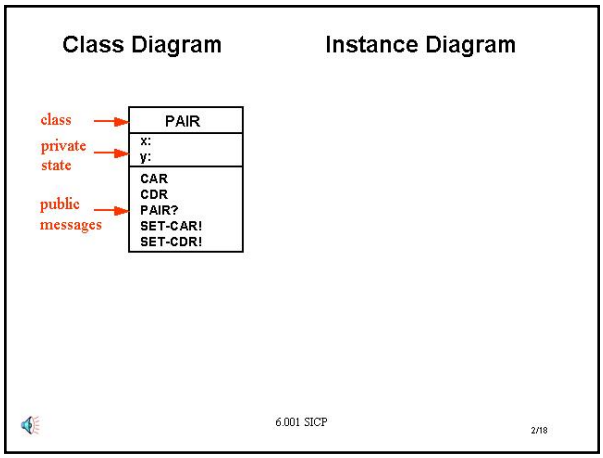
**Slide 13.3.1**




We have seen lots of examples of the first style of programming in the first part of the course. Now we are going to spend some time exploring the second style. What does it mean to create an object-oriented system?

To discuss this, we need some terminology: In an object-oriented system, we will talk about a **class** and an **instance**. A **class** captures a set of objects, with common behavior. For instance, `cons` in our previous example was a class. By convention, to use a class we will have a **maker** procedure that creates **instances** of this class.

An **instance** will be a particular and specific version of a class. For example, `foo` or `bar` in our earlier examples were instances of the `cons` class. An instance takes messages in the manner defined by the **maker** procedure and uses them to manipulate the particular values of the instance. So we expect as a consequence to have lots of instances of a particular class in our system.

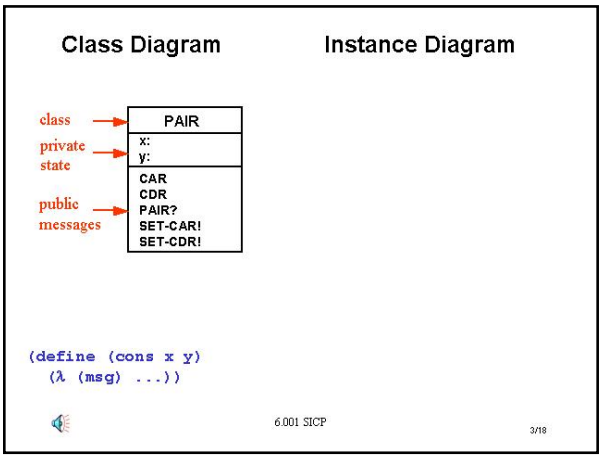


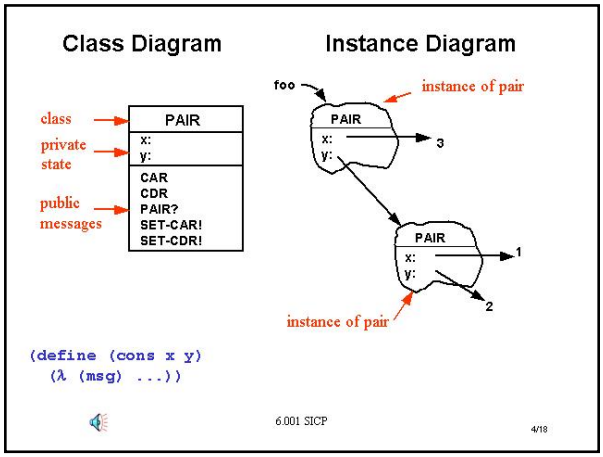

**Slide 13.3.2**

So here is a good way to conceptualize these ideas, and in particular the differences between them. Associated with a class will be a **class diagram**. This contains information such as the name of the class (`pair`), the private state that belongs to the class (`x` and `y`), as well as what public messages are recognized by this class (`car, cdr, pair?, set-car!` and `set-cdr!`). Note that these public messages define the interface to the class of objects. This class diagram thus captures the relevant information about a class.

**Slide 13.3.3**

Thus in our earlier example, `cons` defines a class. Our previous definition for `cons` is our particular way of implementing this class, or constructing elements of this class.





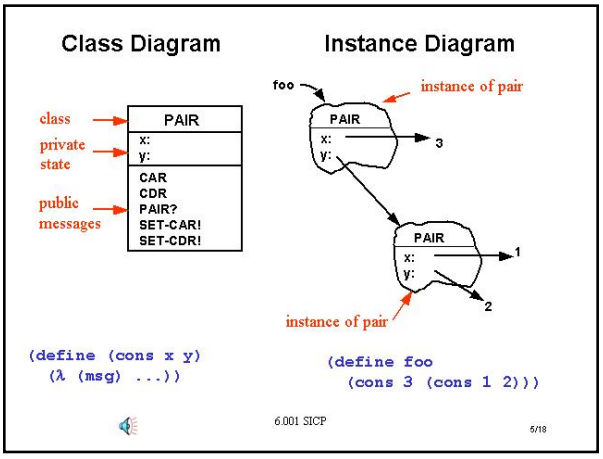

**Slide 13.3.4**

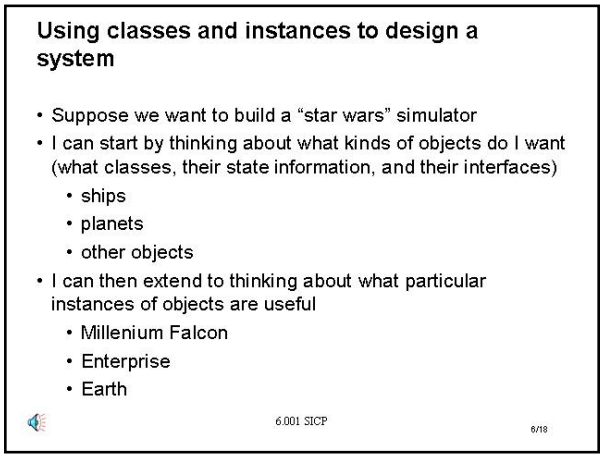
And that leads naturally to the idea of an **instance**. When we use the **maker** associated with this class, we create particular instances of the class, or examples from this class. We can represent diagrammatically in an **instance diagram**. Within that diagram, we will have information about the type of each instance as well as specific values for the internal state of each instance. Note that these values could themselves be other instances of classes.

**Slide 13.3.5**

So if I define the example expression shown, in my instance diagram I create two instances of the class `pair`, both created by the maker procedure `cons`. Within each, I have bindings for, or specific values for, the internal state associated with each instance. Notice how these bindings can be simple values like numbers or pointers to other instances of classes.

So we see that a class defines a set of objects and an instance is a particular version of an object from some class.




**Slide 13.3.6**

So how do I use the ideas of classes and instances to start designing a system? Let's suppose, as an example, that I want to build a simulator for a star wars game: it would have ships that could fly through space, land on planets, shoot other ships.

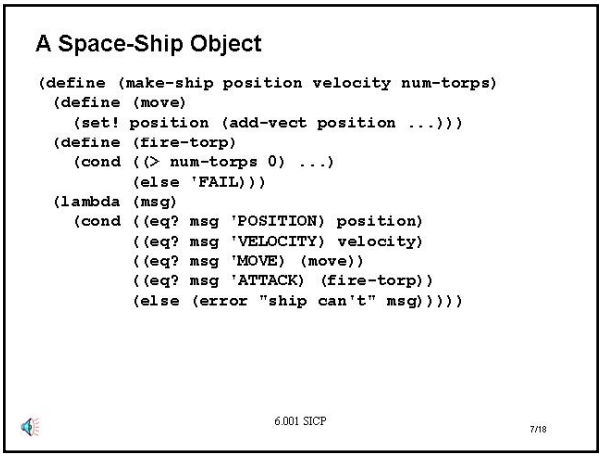
I can start by thinking about what kinds of objects do I need? That will tell me what kinds of class I need and what state information is needed for a class and what interfaces between classes are needed. It might say, for example, that I want some ships. Since ships will need to move, this helps me decide what kind of state a ship will need.

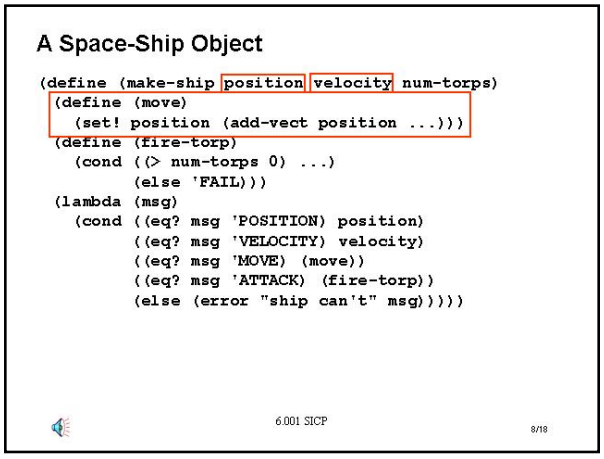
Thus I begin thinking about the system in terms of what kinds of objects and what information associated with those objects do I want. I can then extend this to start thinking about particular instances of objects. How many, what state for each, and so on.

So let's see how the idea of a class, and instances of a class, can be used to design an object-oriented system.

**Slide 13.3.7**

We will use our star wars simulator to explore the use of classes and instances of classes to build object-oriented systems. The first class of objects in my system will be ships. So here is a procedure for making instances of the class of ships. This **maker** procedure defines the actual class.




**Slide 13.3.8**

Given the idea of a ship, I can turn to the question of what behavior I want for instances of that class. Clearly a ship needs to be able to move. Note that this then tells me that I will need some information about where the ship currently lies and how it is moving, as part of the class definition. In this case, I choose to pass that information in, when I actually construct the instance. So this tells me I will need `position, velocity` and maybe some other things as inputs to the class constructor.

Actually, I am cheating here. I know that in order to move, I need to have that kind of information as part of the class, but in fact I could have created a constructor with no parameters, and simply have within it a `let` clause that contained initial default values for the position and velocity.

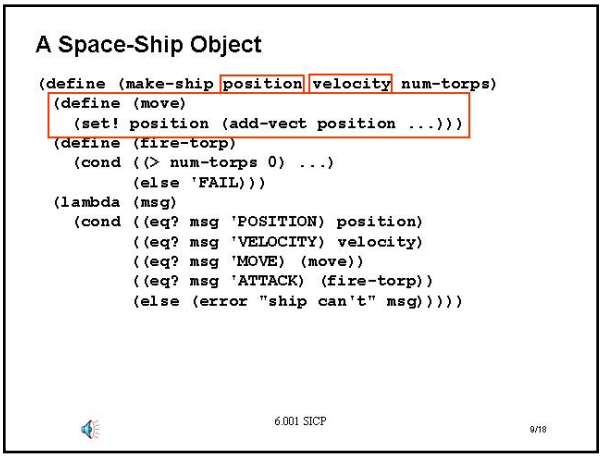
Notice how thinking about what I want my objects to do helps me to decide what information should be captured as local state, and what information should be passed in when I create instances of this class.

**Slide 13.3.9**

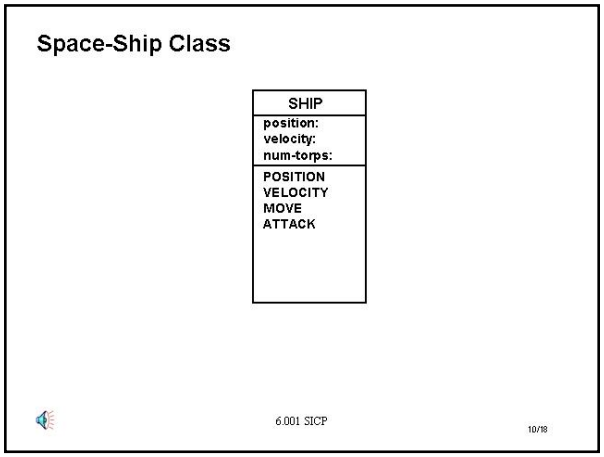
And what about the class itself? When I use this **maker** procedure, it will return an instance of a ship, which will be represented by one of these message-passing `lambdas`.

Note the form. It takes as input a message, and either returns information about the state of the ship, or causes one of the internal procedures to be executed. This looks a lot like the generalized form we used for our `cons` pair earlier.

Similarly, we will have internal procedures for manipulating the data values, very much like our `cons` example. The only other thing to note is how the last clause of our message passing procedure is a bailout clause. It says: if you give me a message that I don't recognize, I'll let you know so that you don't try to do something you can't.

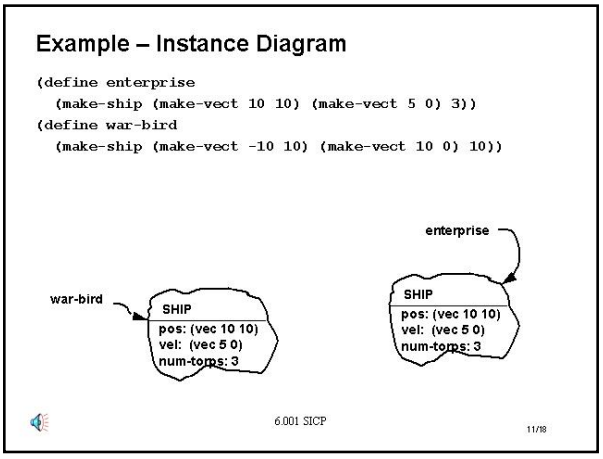


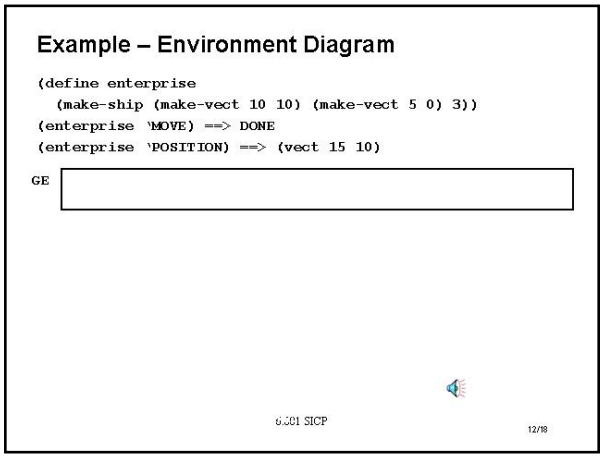

Thus here is a definition of a class: a **maker** procedure that creates ships.




**Slide 13.3.10**
So let's gather that together in our class diagram. Here is the class diagram of a ship, based on that definition, including state variables, and messages for manipulating those state variables. Those messages now define the interface to instances of this class.

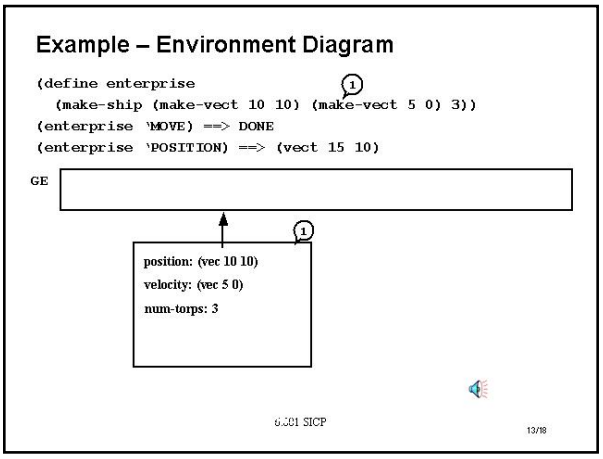
**Slide 13.3.11**
Now we can make some instances, as shown. Of course, we are assuming some abstraction for vectors, which we could easily define. Note that by using the class's **maker** procedure, I have created different instances, each with its own state variables and its own procedures for manipulating that state information. The interfaces to these instances are defined as the set of interfaces from the class definition. Each of these objects will accept the same set of messages, but case their **own** internal state to change.

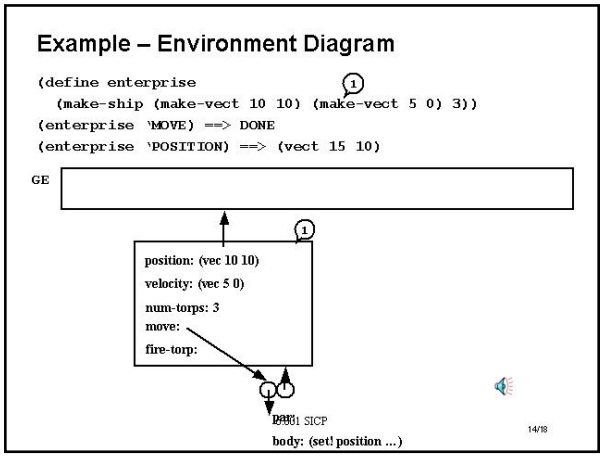



**Slide 13.3.12**
To see this, let's use our environment diagram to see how our simulation evolves. We want to see both how instances are generically maintained in the system, as well as examining the details of our particular implementation. Let's evaluate these three expressions in order, to see how our classes create instances, and how those instances keep track of state information, as our simulation moves forward.

**Slide 13.3.13**
First, we will use our `make-ship` procedure to create an instance of the class. Evaluating that first expression creates a frame, through the application of the procedure, in which the formal parameters are bound to the values of the arguments passed in.



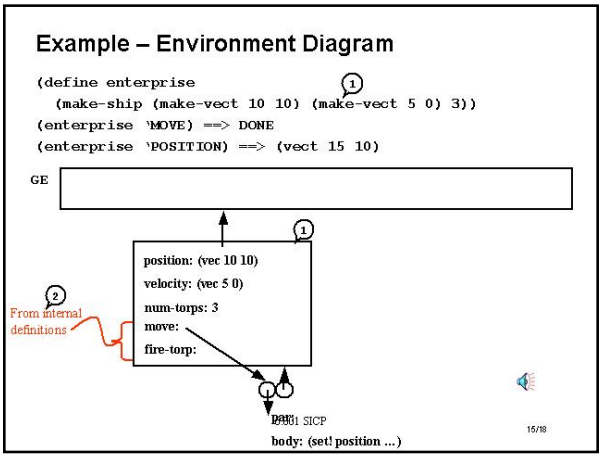


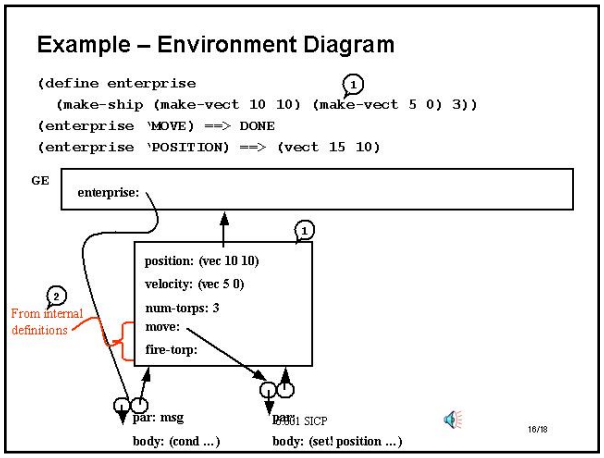

**Slide 13.3.14**

Having created that frame, we then evaluate the body of the procedure with respect to it. Note what that does. The internal definitions cause a binding of a variable in this frame to the value shown, in particular, `move` gets bound to a procedure whose environment pointer points to this frame. Thus, we have an internal procedure, which will have access to the local state variables captured in this frame.

**Slide 13.3.15**

Thus, those two internal `defines` create bindings for names in that frame that point to local procedures with access to the bindings of this frame. Thus this frame contains information about this instance of the class, the state variables and the local methods that will be used to change those variables.



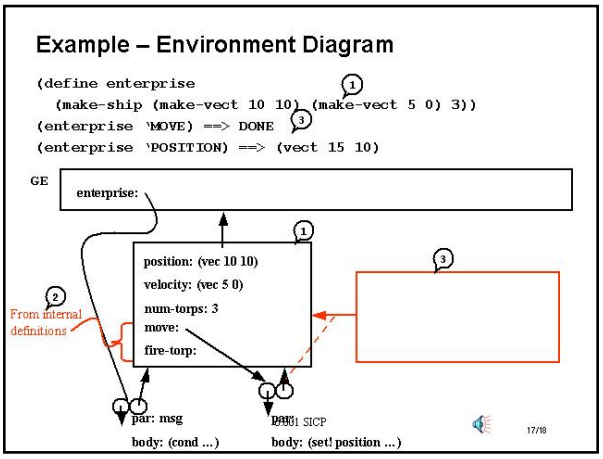



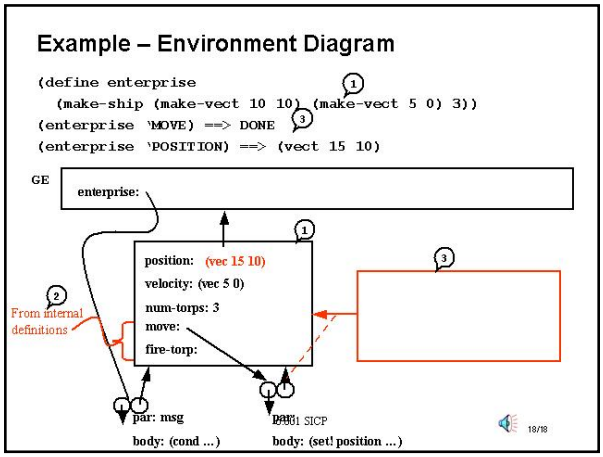

**Slide 13.3.16**

Having evaluated the local `defines`, we then evaluate the rest of the body of this constructor, `make-ship`. That evaluates that last `lambda`, creating a procedure object whose environment pointer also points to this frame, and a binding for the name in the global environment that points to this procedure object. Again, we get a message-passing object with local state and methods for changing the local state.

**Slide 13.3.17**

Suppose we ask this particular instance of a ship to move. This says to send the procedure representing this ship the message `move`, and we know that this reduces to evaluating the actual procedure `move` with respect to this frame. Applying that procedure creates a frame, even though there are no arguments to bind, and relative to this frame we evaluate the body of `move`. This then says to change the position.






**Slide 13.3.18**

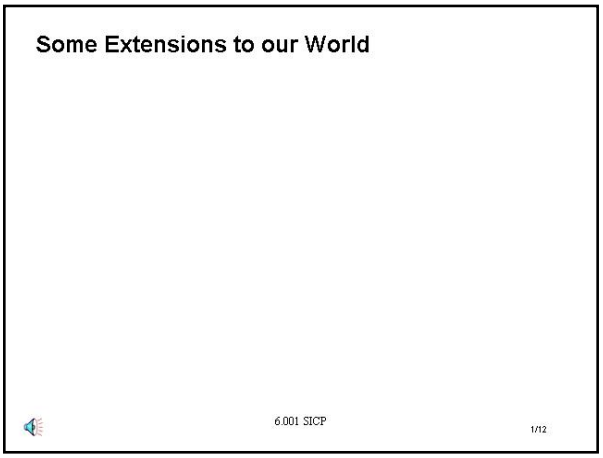
And of course evaluating the body of this procedure causes us to mutate the current value for `position`. Just as we saw with our `cons` example, we have changed the state of our instance. Thus, if we ask for the position of our object now, we will get this new value as the answer.

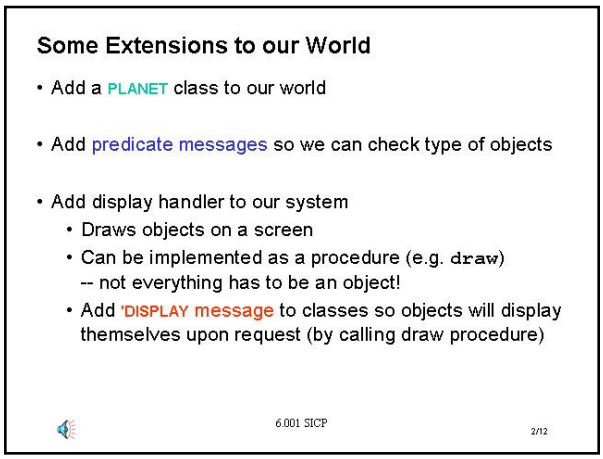
Thus, our local instance of a ship captures within it state information and procedures to change **that** state information. The actual instance object is a procedure that uses messages to determine what to do.

# 6.001 Notes: Section 13.4

**Slide 13.4.1**

The key thing to note is how we can use objects to modularize our design. Notice in the previous case that we can leave some methods blank and still test our system, as we can just fill these methods in when we need them. Also notice how we use the idea of what things we wanted the object to be able to do to define what state information we needed, and what methods we needed to deal with that information. So let's build on this idea to see what other capabilities we can add, and how thinking about things in terms of classes of objects and instances of those classes allows us to easily design interactive systems.




**Slide 13.4.2**

So what kinds of things could we add to our system?

First, we could add new classes, for example a **planet** class, which should have a different behavior from ships.

Second, we forgot to allow for types and tags in our system. This means we need to add **predicate** messages and methods so that objects can identify their type. Note that this arises once I add more than one kind of object to my system (e.g. with just ships I might not need it, but once I add planets I have to distinguish between types of objects).

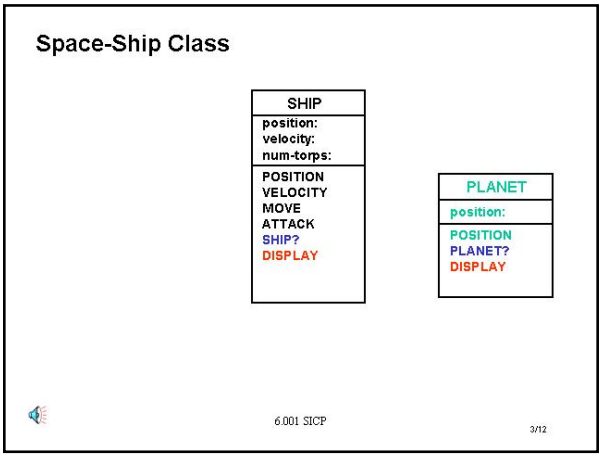
And we might want a display handler, something that draws the position of my objects on a screen. This we will see can be implemented as a procedure, so that not everything needs to be an object in our system. To do this, we will need to modify our classes so that every object can display itself on demand.

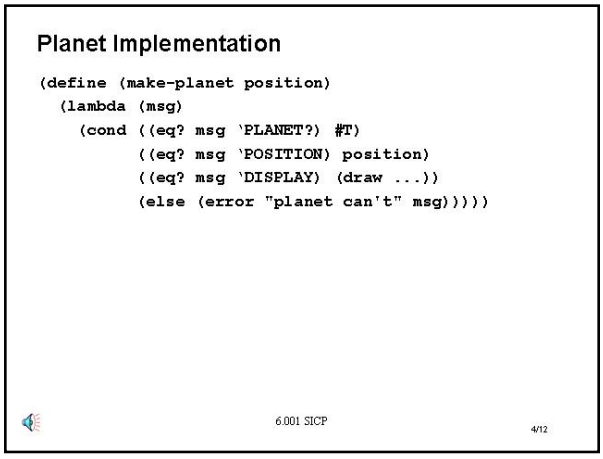
With this framework for modifying our system, let's see how one goes about extending object-oriented systems.

**Slide 13.4.3**

If we add planets to our system, we will have a new class diagram. In addition to our earlier class, we now have a class for planets, with some state information and some methods for manipulating that information. Notice that as part of our design we add two new things to both this class and our original ship class: a predicate for testing types, and a method for handling display.

Now watch how we can add new instances to our system and how we can easily return to our original design and modify it to add new components and methods.




**Slide 13.4.4**

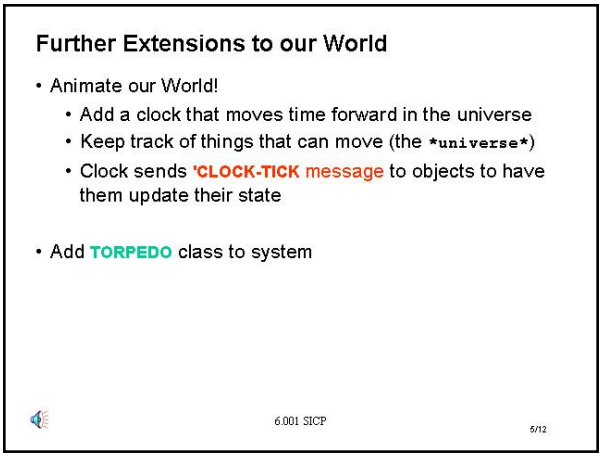
To start, here is our **maker** procedure for the planet class. Note that the only local information is `position` and the instances will recognize three methods. Here is our first version of a predicate: the message `planet?` will return the true value, to indicate that this object is of that type, hence we have a new way of tagging objects. And notice how we can add methods that simply execute a procedure without returning a value. Here `draw` is used for the side effect of graphically displaying the planet.

Note in particular how `planet?` is defining the equivalent of a type tag.

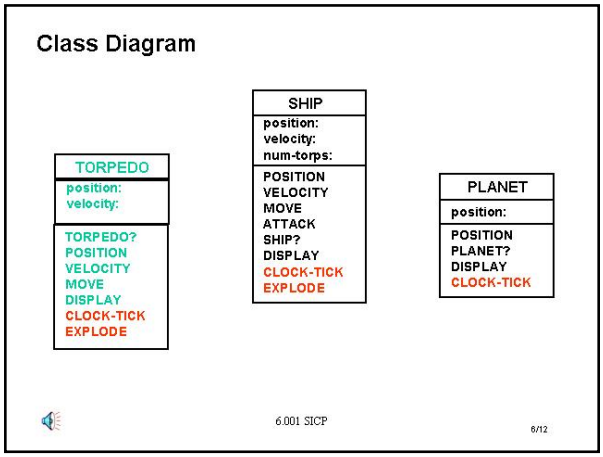
**Slide 13.4.5**

Now that we have seen some simple extensions to our world, what else can we add? Well, the original motivation for building this system is to enable large systems of different kinds of objects to interact. We would like to set up our system so that we can create a bunch of objects, initialize their state, and then let the objects interact in a simulation of some dynamic evolution.

To do this, we just add one new object: a **clock**. If we then keep track of everything in our universe of instances, we can have the clock synchronize the evolution of each object by sending it a message to update its state, which it will do through its own local procedure.




And to add some spice to our system, we will let ships shoot at each other, so we will add a new class, a **torpedo**.



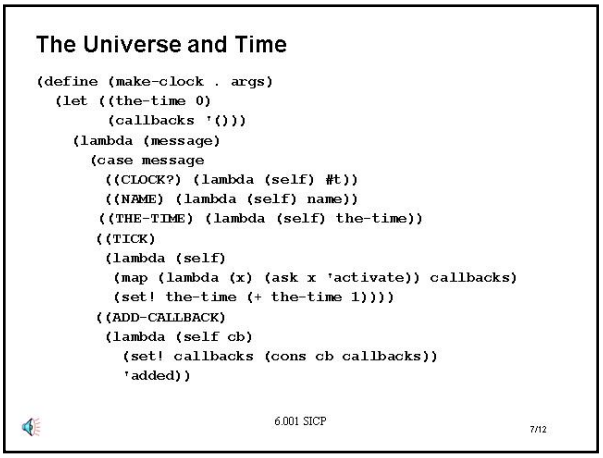

**Slide 13.4.6**

If we add **torpedoes** we need to update our class diagram, and here it is. Notice that this class shares a lot of common information with ships, a point to which we will return later. The top level point to note is how our design is evolving. We started with a single class, and have now added new classes as our desires for the system expand. Each class has a set of local state variables, and methods that it is capable of handling.

The other change to our class diagram is the inclusion of a new method for each object, the ability to accept a synchronization signal from the clock, and then update state.

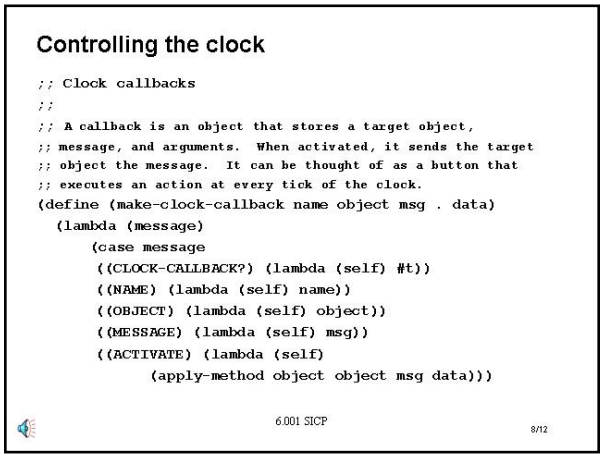
Again, note how our design is evolving. We can add new classes, and we can extend existing classes by adding new methods as our desired behaviors for the system demand them.

**Slide 13.4.7**




We need a way of coordinating the actions and interactions of objects in our universe. One way to do this is to use a clock, which could send a signal to each object to synchronize the passage of time. For example, a clock **tick** could cause a ship to move a small amount, or fire a torpedo, or launch a shuttle. There are several ways to accomplish this; here is a sketch of one. Our clock maintains a list of things to do on each tick of the clock. In particular, it does this by storing, as internal state, a list of **callbacks**, that is, messages to send to specific objects to execute specific actions.

We can then simulate our world by running the clock through a sequence of ticks. On each tick, the clock walks down its list of callbacks, and asks each to **activate.** Intuitively, this means that it asks a set of objects to pass messages to target objects, like our ships.




**Slide 13.4.8**

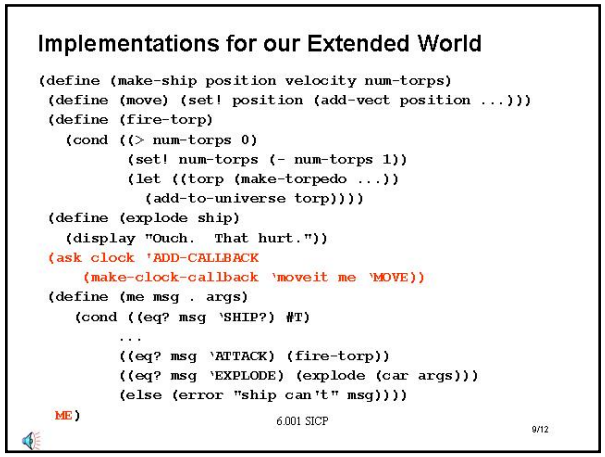
To make this happen, we simply need to create a new object, a **callback.** This is simply a new object that stores a target object, a message and a set of arguments. When activated (by the clock), it sends the target object the message, which will cause something to happen in the world. It can be thought of as a button that executes an action every tick of the clock. The details are not crucial, what matters is the idea that a single object, a **clock**, can control another set of objects, the **callbacks**, that synchronize the actions of objects in the universe.

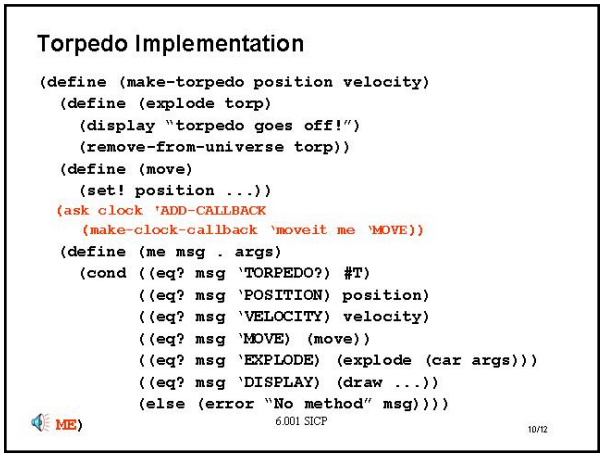
**Slide 13.4.9**

We can also evolve our class definitions. Given that we want to add some new capabilities to our ships, we return to the **maker** procedure and change it. For example, we can add a method to fire torpedoes, together with methods that update the state associated with torpedoes. Also note as part of this how we use a maker to create a torpedo, and adding the torpedo to the universe will presumably cause the torpedo to send a callback message to the clock so that the clock can keep track of it.

We also need to allow ships to explode, and notice the form here. The actual procedure `explode` takes as argument a ship, which needs to be removed from the universe. Notice how to support this we have changed the definition of our message passing object to use that "dotted" argument notation, so that we can pass in arbitrary numbers of arguments to an object. We use that here to allow for the fact that if something collides in our universe, we ask the object to "explode" and we pass in the pointer to the object itself. Thus, in the method, we can then remove that object from the universe (if it has exploded it shouldn't stay around for further simulation!). Thus we have a way of passing objects as arguments to other objects.

Finally, note how we send a callback object to the clock, which will create a message to be sent to this object on each clock tick, in this case, just asking this ship object to **move.**



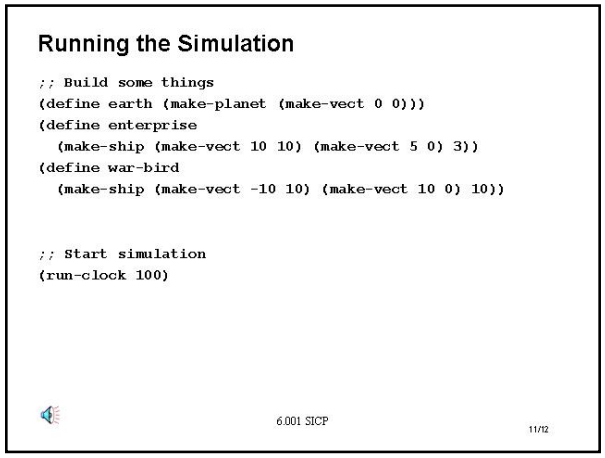



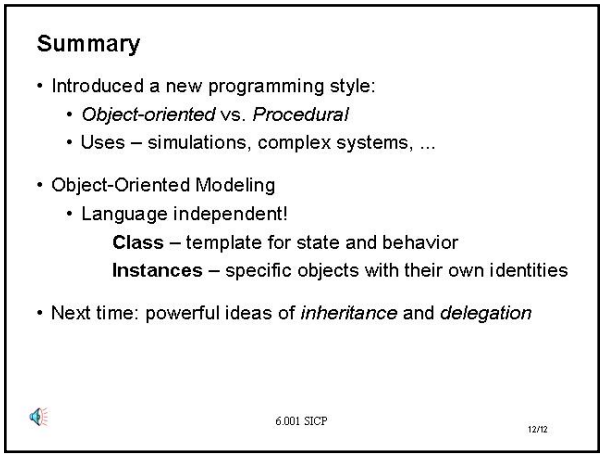

**Slide 13.4.10**

So we are almost done. We need to have something that makes instances of torpedoes. Note that it has a lot of the same form as a ship, but has its own internal procedures for exploding and for moving.

**Slide 13.4.11**

Having done all of this, we can now run a little simulation. We create some instances of objects, we add them to this universe, and then just run the clock to start the simulation.






**Slide 13.4.12**

So here are the key messages to take away from this exercise.