Lecture 20: Dependency Mechanisms

- **Dependency**
  - Objects depend on the state of other objects.
  - For example, we have a lamp with a light switch and a light bulb
    - When the state of the switch is changed, the light bulb is notified of the change
    - This does not imply when the light bulb status changes (burns out) that the light switch is notified
  - There is no explicit relationship between two objects so we must find a way of connecting them. Through a special connection, one object is said to be "dependent" on the other object. Smalltalk has a dependency mechanism to handle the connections between two objects.
- Use collections to store groups of dependent objects.
  - Instead of a light switch, we now have a traffic light with three light bulbs. Each light has to notify all other lights when it turns itself on so they will turn themselves off. In this example, no two lights should be on at the same time.
  - The protocol for sending messages and updating is provided by class Object.
    - A Object sends itself a `changed` message & its dependents are informed automatically via the `update: aMessage,` where `aMessage` can be any message
      - `self changed`
        - dependents determine what was changed
      - `self changed: anAspect`
        - Object informs dependents of a change involving `anAspect`
      - `self changed: anAspect with: aValue`
      - `update: with: from:`
        - Method is inherited from class Object, but it is usually overridden.
  - Methods for adding and removing dependencies
    - `addDependent:`
      - adds the dependency of the argument's object to the receiver's object
      - `tire: addDependent: automobile.` The automobile is sent a message if the state of the tire is changed.
    - `removeDependent:`
      - removes the dependency of the receiver from the argument
      - the receiver no longer updates the argument
      - `tire: removeDependent: automobile.` The automobile is now not updated when the state of the tire changes
    - `release`
      - Releases all of the dependents of an object
    - `dependents`
      - Returns an Ordered Collection containing the dependents
  - Now we can look at the code for the lamp
    - To allow for the light bulb to be easily changed, we'll give it a function `update: signal`
    - The lamp object should also provide the methods for the following
      - Getting and setting its state
      - Showing the state by writing to the Transcript
      - Getting and setting its identification number (id)

```
Object subclass: #Lamp
       instanceVariableNames: 'state id'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Examples-Lamps'!


state
       "Returns the state of the lamp: 0=off, 1=on."
```

```
        ^state.!

state: anInteger
        "Sets the state of the lamp: 0=off, 1=on.
        The self changed message tells the lamps
        dependents it is now on."

        anInteger isZero ifFalse: [ state := 1.
                                    self changed: #on]
                                    ifTrue: [ state := 0 ].! !
id: anInteger
        "Sets the id of the lamp"
        id := anInteger.!

id
        "Gets the id of the lamp"
        ^id.

showState
        "Shows the state of the lamp in the transcript"
        Transcript show: 'Lamp '; show: (id printString).
        Transcript show: ' state: '; show: (state printString); cr.

!Lamp methodsFor: 'updating'!

update: signal
        "If some other lamp has turned on, turn myself off."

        (signal = #on)
        ifTrue: [state := 0.
                    self changed].! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Lamp class
        instanceVariableNames: ''!


!Lamp class methodsFor: 'instance creation'!

new
        "Gets a new instance."

        ^super new! !
```

- Now we will look at the code for the lamplist and traffic signal. Keep in mind that the LampList and the Lamps are the objects doing the work, and the TrafficSignal provides a useful example of a device which could use the LampList.

```
OrderedCollection variableSubclass: #LampList
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Lamps'!


"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

LampList class
        instanceVariableNames: ''!


!LampList class methodsFor: 'instance creation'!
```

```
new
       "Creates a new instance."

       ^super new!

new: size
       "Creates a new instance."

       ^(super new: size)! !

"-------------------------------------------"!

Object subclass: #TrafficLight
       instanceVariableNames: 'lamplist '
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Examples-Lamps'!
```

- In the method `initialize`, each lamp is created, and added to the lamplist. Each lamp in the lamp list is then adds the other lamps to its list of dependents.

```
!TrafficLight methodsFor: 'initialization'!

initialize
       "Creates the three lights and turns the first on"

       | lamp index|
       lamplist := LampList new:3.
       Index := 1.
       3 timesRepeat:
                    [ lamp := Lamp new.
                      lamp state: 0.
                      lamp id: index.
                      lamplist add: lamp.
       (lamplist at: 1) state: 1.
       1 to: lamplist size do: [ :l |
              1 to: lamplist size do: [ :dep |
                     l = dep ifFalse: [
                            (lamplist at: l) addDependent:
                            (lamplist at: dep)]]].! !

!TrafficLight methodsFor: 'accessing'!

lamplist
       "returns the lamplist"
       ^lamplist.!
```

- The method gets the lamp that is on, then turns on the next lamp. If the third lamp was on, then the first lamp is turned on. Each time the light is changed, a message is written to the transcript to log that the light was changed.

```
changeLight
       "advances to the next light in the list"
       | index |
       Transcript show: 'Changing the Lights'; cr.
       index := (self lightIsOn) id.
       (lamplist at: ((index rem: 3) + 1) state: 1.
```

- The method `lightIsOn` is used to tell the traffic light which lamp is currently turned on. It finds the first lamp which is on (only 1 should be on), and returns that light.

```
lightIsOn
	"returns the index of the light that is on."

	^(lamplist detect:
		[ :lamp | lamp state = 1]).

showStates
	lamplist do: [ :lamp| lamp showState].

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

TrafficLight class
	instanceVariableNames: ''!


!TrafficLight class methodsFor: 'instance creation'!

new
	"Creates a new instance and initializes the lights."

	^(super new) initialize.! !
```

- Now we can call the method `showStates` to reveal the light that is on. Lamp 2 is initially set on, then the `changeLight` message is sent to `aTrafficLight`. Notice how Lamp 2 automatically turns itself off when `changeLight` turns on Lamp 3. This kind of automation is the primary value of the dependency mechanism.

```
| aTrafficLight |
	aTrafficLight := TrafficLight new.
	((aTrafficLight lamplist) at: 2 ) state: 1.
	aTrafficLight showStates.
	aTrafficLight changeLight.
	aTrafficLight showStates.
```

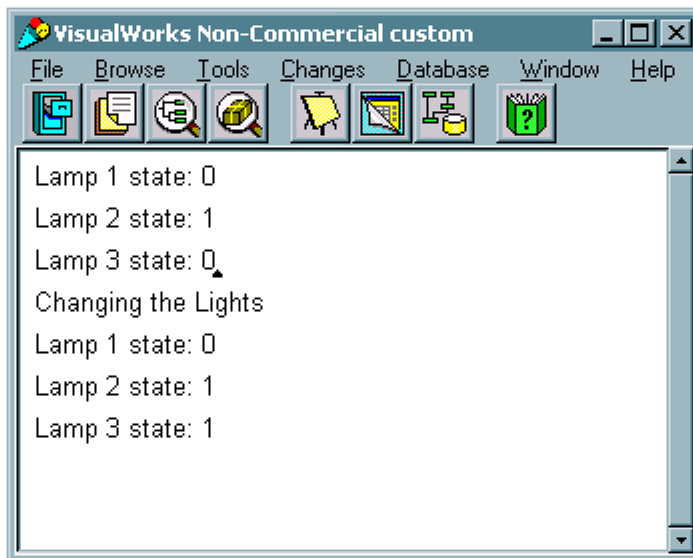
VisualWorks Non-Commercial custom

File Browse Tools Changes Database Window Help

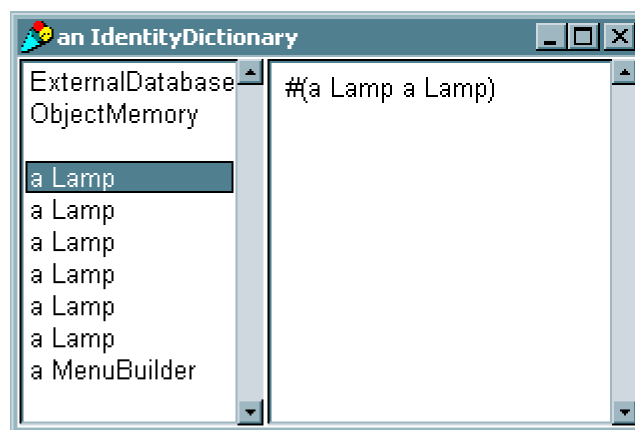
Lamp 1 state: 0
Lamp 2 state: 1
Lamp 3 state: 0
Changing the Lights
Lamp 1 state: 0
Lamp 2 state: 0
Lamp 3 state: 1

- What happens if we break the dependency on one of the lamp? Let us remove the 2nd lamp's dependency on 3rd lamp from the ordered collection. Notice that when the 3rd lamp turns on the 2nd lamp never turns off.

```
| aTrafficLight |
aTrafficLight := TrafficLight new.
((aTrafficLight lamplist) at: 2 ) state: 1.
aTrafficLight showStates.
(aTrafficLight lamplist at: 3) removeDependent:
       (aTrafficLight lamplist at: 2).
aTrafficLight changeLight.
aTrafficLight showStates.
```



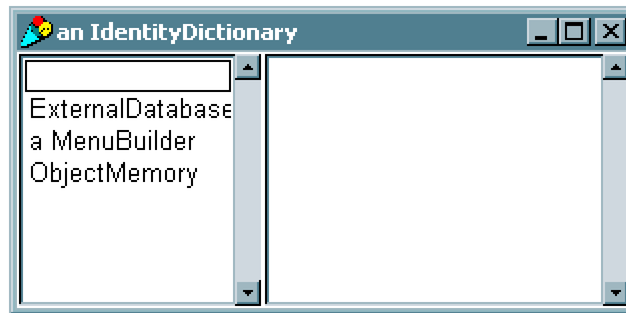
- For an arbitrary object, Smalltalk does not remove the dependents from the object when it is no longer in use.
- The system wide dependencies are stored in an identity dictionary `DependentsFields`. Upon inspection of this dictionary, we can see the dependencies we have created in the last two examples. Each lamp has an entry that includes each of the other lamps in the traffic light. Note that there are six lamps, as the two examples each added 3 lamps.



- Because we do not have a direct way to access the lamp objects from the previous examples, we will remove their dependencies by removing their entries in

DependentsFields. The following code will remove all keys of the object class Lamp. The code simply creates a collection of keys to be removed, then removes each one.

```
| keys |
keys := OrderedCollection new.
DependentsFields associationsDo: [ :anObject |
       ((anObject key) isKindOf: Lamp)
              ifTrue: [ keys add: (anObject key) ]].
                keys do: [ :aKey |
                Transcript show: 'Removing ', aKey printString; cr.
                DependentsFields removeKey: aKey ifAbsent: []].
```



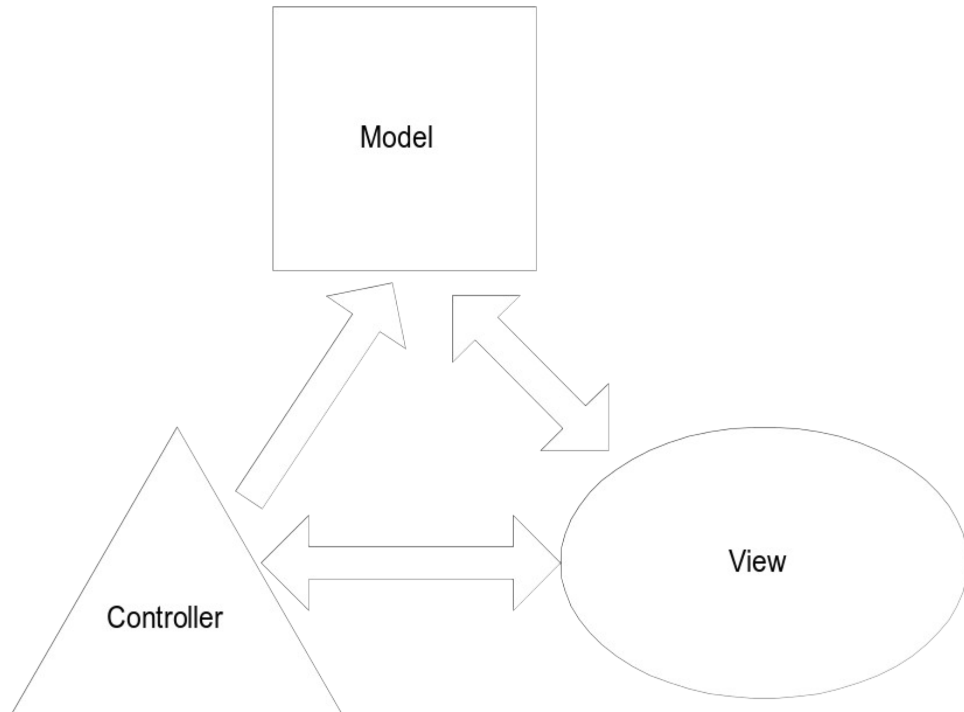
- To avoid leaving stray dependencies in DependentsFields, we need to add a method to TrafficLight that will remove the dependencies of the Lamps. This is done by simply sending the release message to each Lamp.

```
removeDependents
       "Removes the dependents of each lamp in the TrafficLight"

       lamplist do: [:lamp | lamp release ].
```

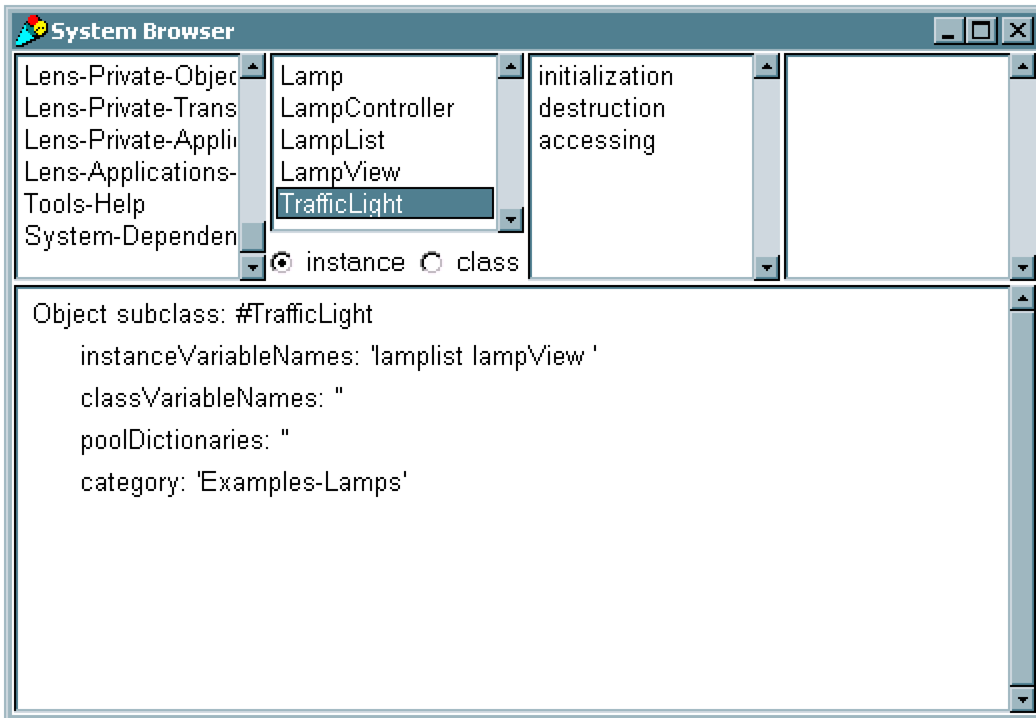
- After the last showStates message, we should now add
  aTrafficLight removeDependents.
- Inspection of DepedentsFields shows that we have removed all of the dependencies.

Lecture 21: The Model-View-Controller Paradigm

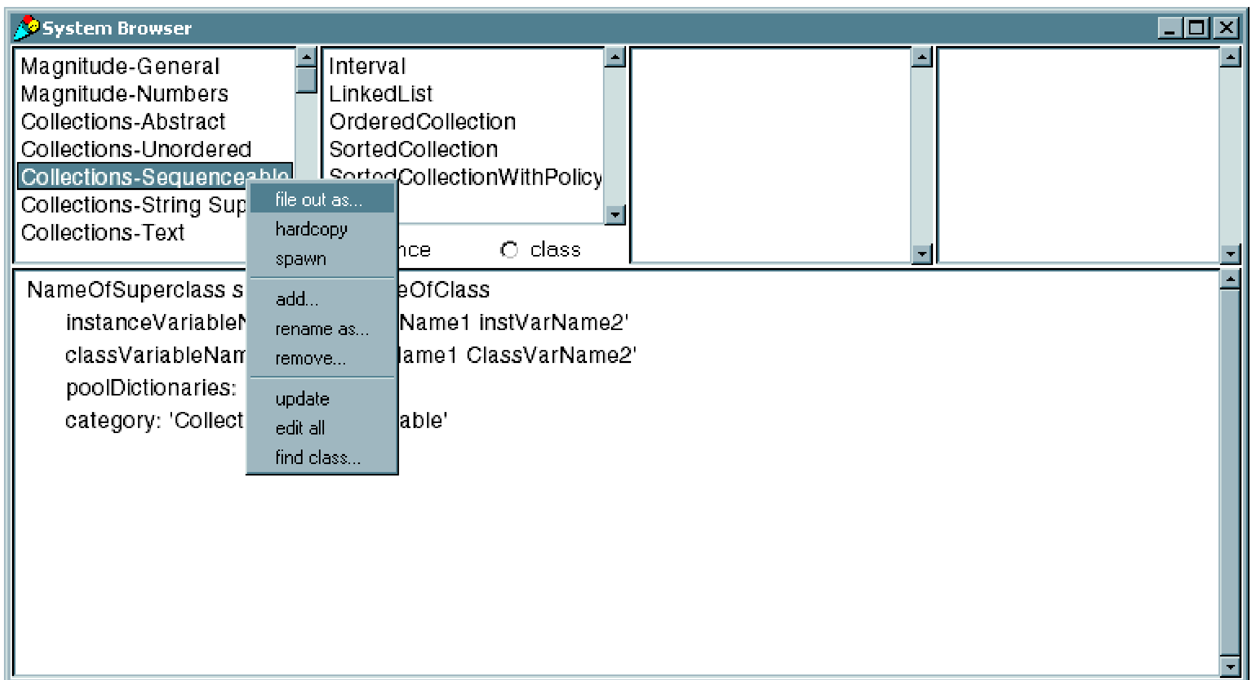
- **Definitions**
  - Model: The object to be looked at and/or modified
    - Provides the details to be displayed
  - View: The object that determines the precise manner in which the model is to be displayed (i.e. a window manager)
    - Displays the model and provides visual feedback for controller interactions
  - Controller: The object that handles the keyboard and mouse interactions for this view
  - The MVC Triad

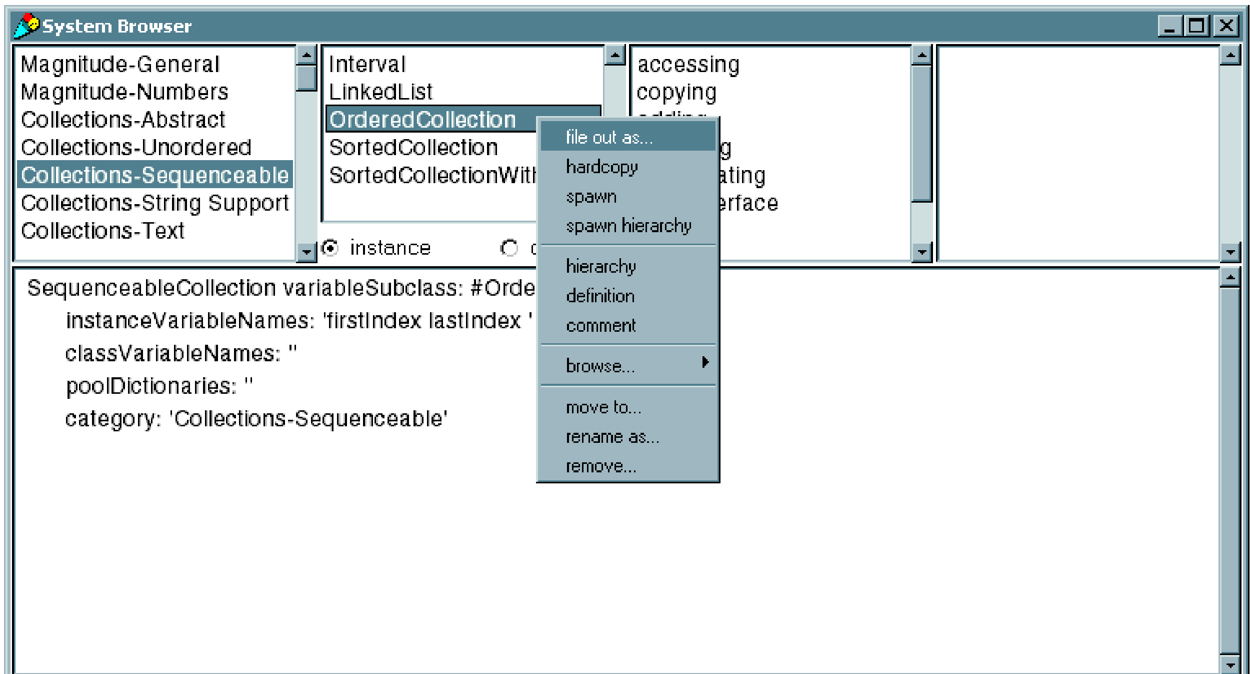


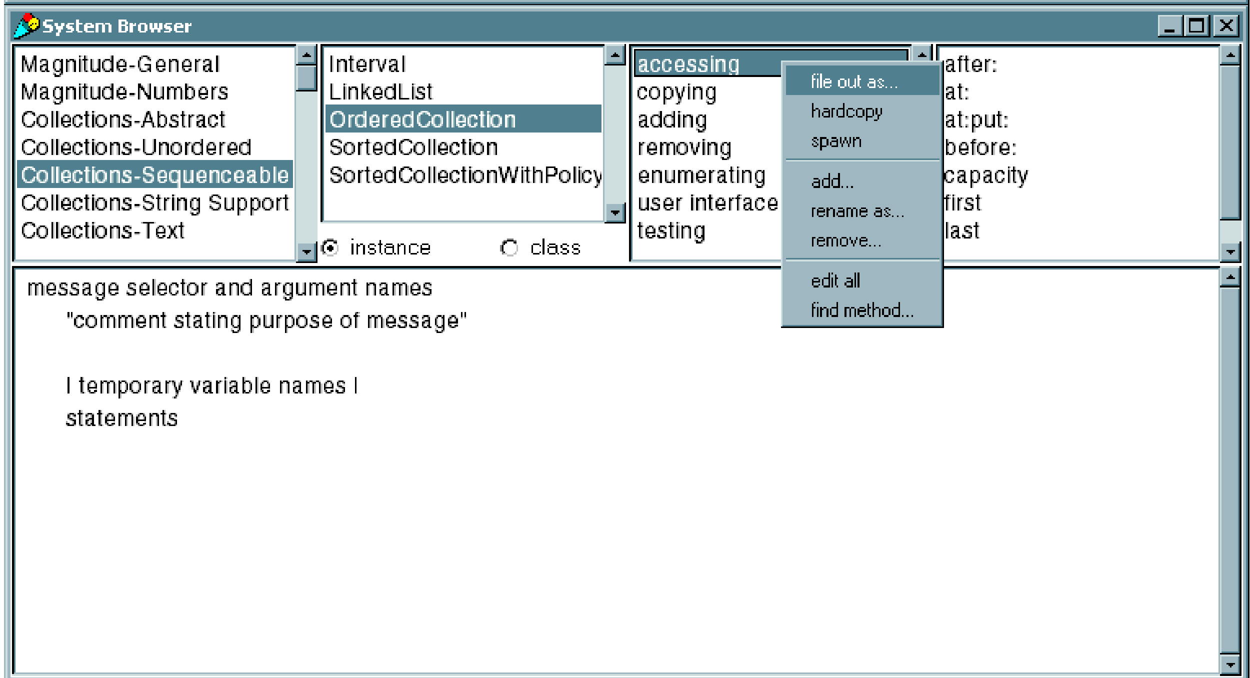
- The view and the controller interact to provide a graphical interface to the model.
  - An example of a MVC application is the browser. The browser is composed of 5 Views, each with its own controllers. The model contains the entire source, and the views and controllers interact to display the source code.

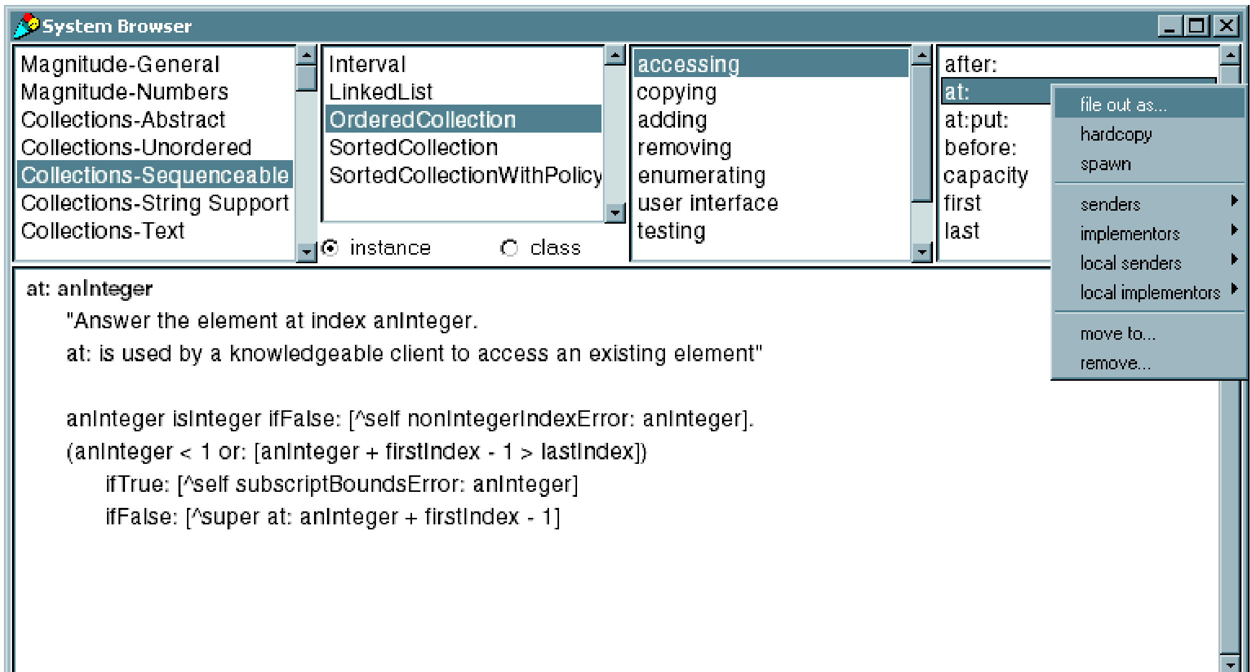


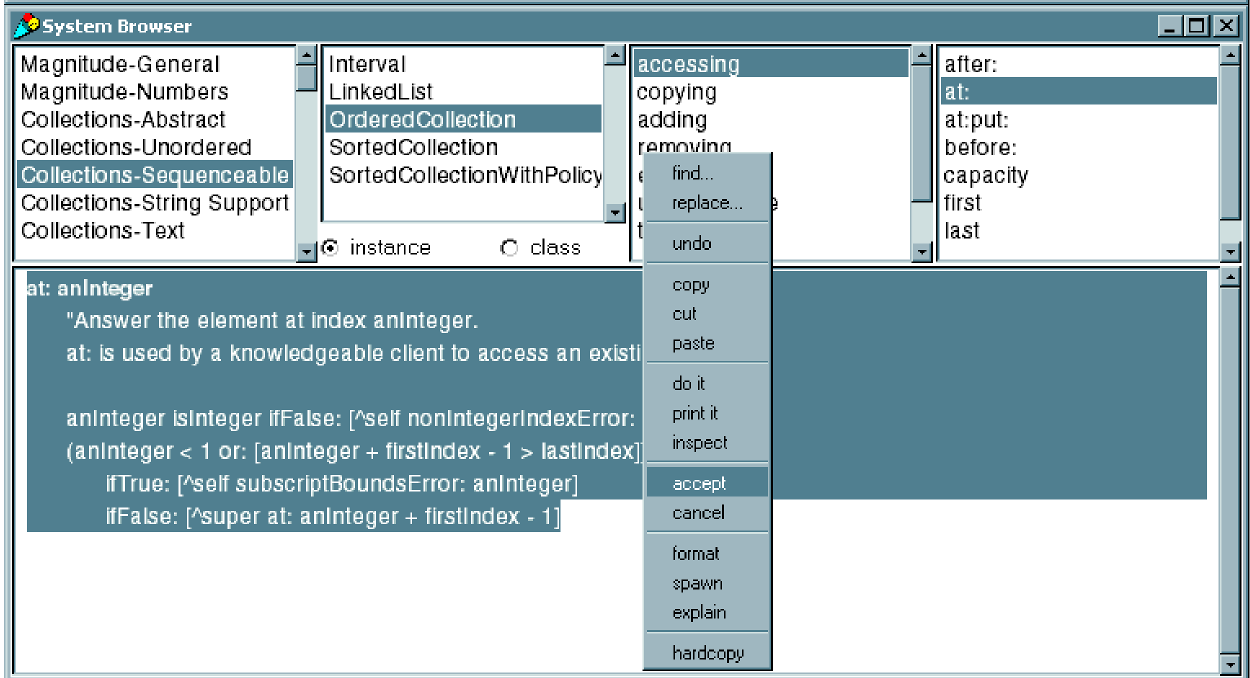
- Each panel of the browser has its own controller, notice how a right mouse button's menu is different in each panel. Its also important to note that each time a item in the SelectionView is clicked one, the other views change as well.




System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
instance
accessing
copying
file out as...
hardcopy
spawn
spawn hierarchy
hierarchy
definition
comment
browse...
move to...
rename as...
remove...
SequenceableCollection variableSubclass: #Orde
instanceVariableNames: 'firstIndex lastIndex '
classVariableNames: ''
poolDictionaries: ''
category: 'Collections-Sequenceable'


System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
SortedCollectionWithPolicy
instance
class
accessing
copying
adding
removing
enumerating
user interface
testing
file out as...
hardcopy
spawn
add...
rename as...
remove...
edit all
find method...
after:
at:
at:put:
before:
capacity
first
last
message selector and argument names
"comment stating purpose of message"
| temporary variable names |
statements


System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
SortedCollectionWithPolicy
instance
class
accessing
copying
adding
removing
enumerating
user interface
testing
after:
at:
at:put:
before:
capacity
first
last
file out as...
hardcopy
spawn
senders
implementors
local senders
local implementors
move to...
remove...
at: anInteger
"Answer the element at index anInteger.
at: is used by a knowledgeable client to access an existing element"
anInteger isInteger ifFalse: [^self nonIntegerIndexError: anInteger].
(anInteger < 1 or: [anInteger + firstIndex - 1 > lastIndex])
ifTrue: [^self subscriptBoundsError: anInteger]
ifFalse: [^super at: anInteger + firstIndex - 1]


System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
SortedCollectionWithPolicy
instance
class
accessing
copying
adding
after:
at:
at:put:
before:
capacity
first
last
find...
replace...
undo
copy
cut
paste
do it
print it
inspect
accept
cancel
format
spawn
explain
hardcopy
at: anInteger
"Answer the element at index anInteger.
anInteger isInteger ifFalse: [^self nonIntegerIndexError:
ifTrue: [^self subscriptBoundsError: anInteger]
ifFalse: [^super at: anInteger + firstIndex - 1]

- **Model**
  - While the Object class handles dependency coordination, as seen in the TrafficLight/LampList example, most model objects are created as a subclass of Model.
  - Object vs. Model
    - Object uses a global dictionary to store dependents.
  - This approach provides global dependency coordination, but dependents must be explicitly removed.
    - Model holds the collection of dependents in an instance variable
  - The model is able to find the dependents faster, hence the methods involving the dependents is speeded up.
  - Failure to release an object can be safely ignored. Garbage collection is able to remove obsolete dependents.
  - The traffic light example presented in the previous lectures is an excellent example. To simplify the model, we will now focus on the `LampList` and `Lamp` classes.
  - To gain the features of Model, the `Lamp` and `LampList` classes should now be subclassed off `Model`, rather than `Object` or `OrderedCollection`. It is important to note that two instance variables have been added to `LampList`: `numin` and `theList`.
    - `numin` is an internal counter, which will be discussed when its implementation is shown.
    - `theList` is an `OrderedCollection` used to store the list of Lamps, since `LampList` is no longer subclassed off of `OrderedCollection`.

```
Model subclass: #Lamp
       instanceVariableNames: 'state id '
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Examples-Lamps'

Model variableSubclass: #LampList
       instanceVariableNames: 'numin list'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Examples-Lamps'
```

    - Several changes must be made to existing methods, and new methods must be added to compensate for `LampList` no longer being subclassed off of `OrderedCollection`.
  - Initialization must now create `theList` as an `OrderedCollection`.

```
initialize
       "Sets the count to zero."
       theList := OrderedCollection new.
       numin := 0.
```

  - Now that LampList is subclassed off of Model, it is part of the dependency mechanism. To maintain the dependency updating process, LampList must implement the update: method. The instance variable numin is used in this method to count the number of Lamps that have reported in to the LampList object. Once all lamps have reported in, the LampList object can send the changed message.
  - In the next lecture on Views, we use the lamplist as a model for a LampView. We wait for all of the lamps to report in to avoid a race condition where the view is redrawn before all lamps have had a chance to update their state.

```
update: signal
    "Waits for all lamps to report in, then redraws the view."

    numin := numin + 1.
    (numin = self size) ifTrue: [
```

```
                numin := 0.
                self changed.]
```

- Methods to access and add to the LampList must now include implementation of `add:`, `at:`, `do:`, `detect:`, and `size` so other methods will not break. It should be noticed, if other methods were needed, they could be implemented simply by passing the message to `theList`.

```
add: aLamp
       "Adds aLamp to theList."
       theList add: aLamp.
       ^theList.

at: anInteger
       "returns a Lamp for theList."
       ^(theList at: anInteger).

detect: aBlock
       "Passes a detect message to theList"
       ^(theList detect: aBlock).

do: aBlock
       "Tells theList to do aBlock."
       ^(theList do: aBlock).

size
       "Returns the size of theList."
       ^(theList size).
```

- Rather than having the `TrafficLight` create the dependency, the method `make:` now adds each lamp to the `LampList` as a dependent, as well as make each lamp dependent on every other lamp.

```
make: anInteger
       "Makes a lamp list with anInteger number of
       lamps input by the user."

       | lamplist lamp |

       lamplist := LampList new: anInteger.
       anInteger
              timesRepeat:
                     [ lamp := Lamp new.
                       lamp id: (lamplist size + 1).
                       lamp state: 0.
                       lamplist add: lamp.
                       lamp addDependent: lamplist].
       1 to: lamplist size do: [ :l |
              1 to: lamplist size do: [ :dep |
                     l = dep ifFalse: [
                            (lamplist at: l) addDependent:
                                   (lamplist at: dep)]]].
       ^lamplist
```

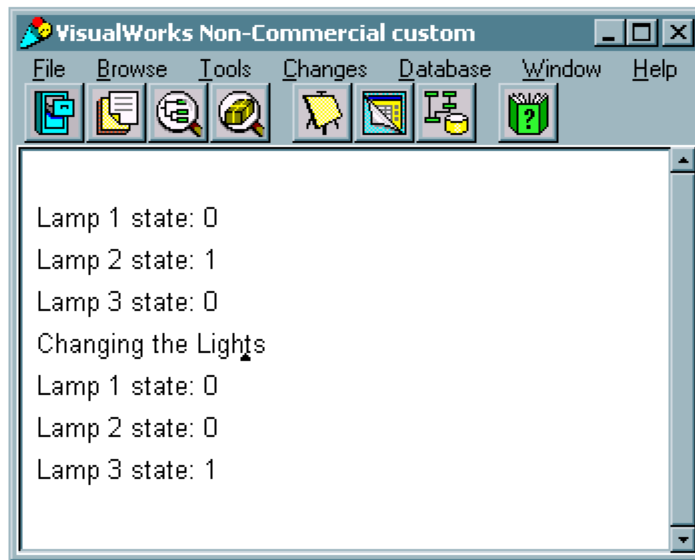
- No changes are needed for accommodating `Lamp's` new subclassing.
- With one modification, the examples used for the `TrafficLight` will work as well now. Since the `make:` method creates all of the dependencies, `TrafficLight's` `initialize` method only has to make the light.

```
initialize
   "Creates the three lights and turns the first on"
```

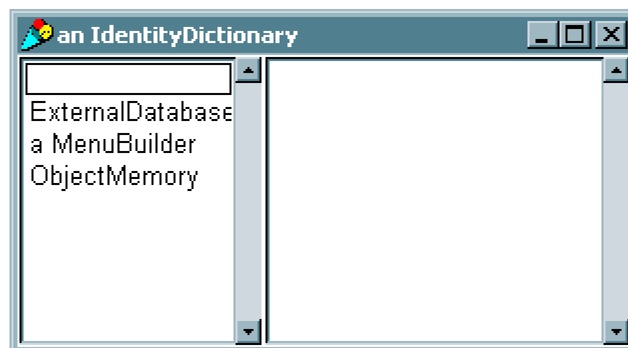
```
lamplist := LampList make:3.
^self.
```

- Now we can look at the same code used in earlier `TrafficLight` examples.

```
| aTrafficLight |
      aTrafficLight := TrafficLight new.
      ((aTrafficLight lamplist) at: 2 ) state: 1.
      aTrafficLight showStates.
      aTrafficLight changeLight.
      aTrafficLight showStates.
      "Note - we no longer need to explicitly
      remove dependents"
```



  - The exact same code used earlier produces the exact same output to the transcript. The only difference can be seen by inspecting `DependentsFields`. Notice that there are no dependents left behind? Since the `Lamp` and `LampList` objects were all subclassed off Model, the dependents were all stored locally in an instance variable, and removed once the object executed.



- Given the simplicity of maintaining dependents, the model concept is used extensively in the following:
  - Dependent Views, which ask for data

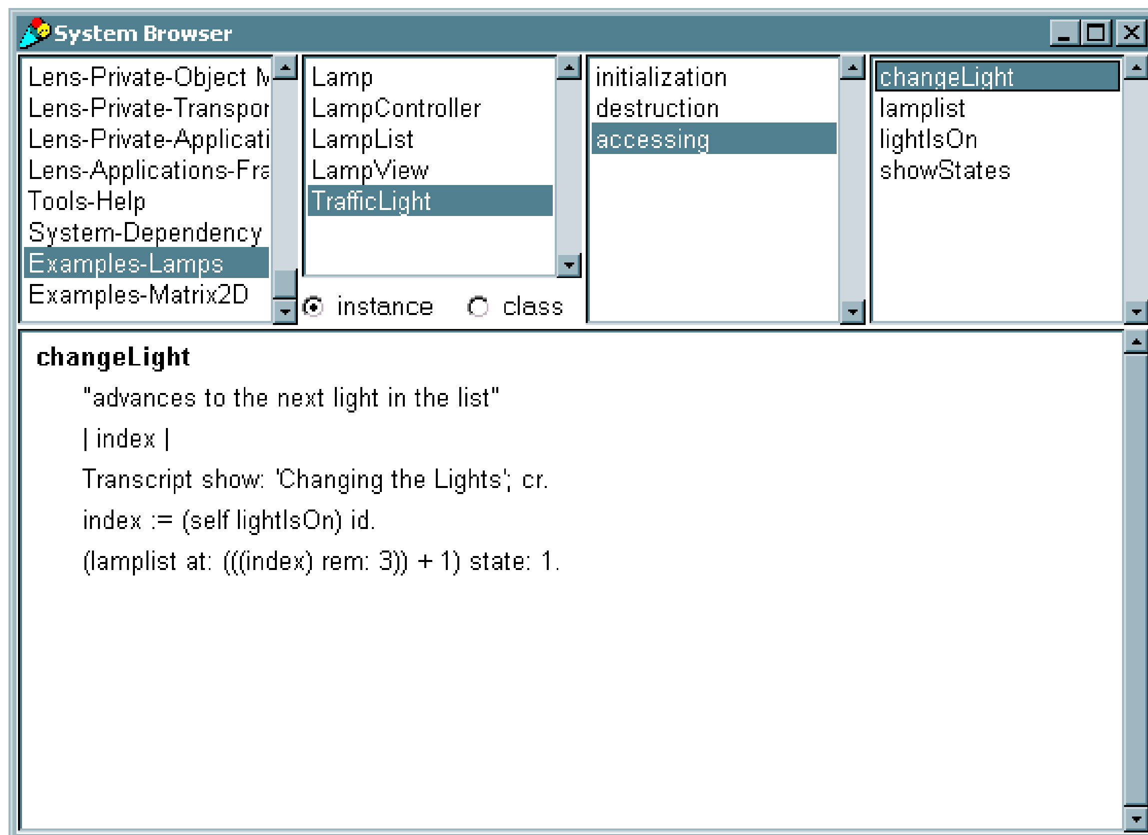
- Example: Real-time graphs and charts, one model could feed two different windows data to be displayed.
    - Controllers associated with dependent views, which supply user input data and request menu operations
- Example: modifying the menu choices so the choices are different depending on what was clicked on
    - Dependent buttons, which request button operations
- Example: Grey'ing out inactive buttons
    - Other models, which request data processing and other services
    - The model itself, which requests data processing and other services.

- We will look at coupling the Model with a View and Controller in the next lectures

.

Lecture 22: The View

- **View**
  - The view is responsible for displaying aspects of the model. Because there are many kinds of models, there are many kinds of views, ranging from very simple to incredibly complex.
  - A view can be thought of as a part of a window in which a visual object is displayed. The object can be passive, such as an image or text, or be an active object that updates itself according to changes in the model, such as a real time graph.
    - The browser window is an excellent example.
  - Each pane is a view. The four top panes are SelectionView objects, and the bottom pane is a TextCollectorView object.



- Every view must implement the following instance methods
  - `displayOn: #anAspect`
- Completely builds the contents of the view
- Called when a view is first created and each time the entire view is redrawn (e.g. uncovered by another window)
  - `update: #anAspect`
- Called whenever the model changes (e.g. sends itself a `changed:` message)
- Used to reconstruct all or portions of a view depending on how the model was changed (indicated by #`anAspect` symbol)
- If desired, #`anApsect` can be ignored in either of these methods.
- Suppose we wish to create a view for the Traffic Light example, we would now implement these methods and a class definition for a new class, `LampView`. The view will be an instance of `AutoScrollingView` with three lamps in it.

- In the class definition, an instance variable must be kept so the view knows what window it is in.

```
AutoScrollingView subclass: #LampView
   instanceVariableNames: 'window '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples-Lamps'
```

- For now, update: needs to only re-display the view.

```
update: aModel
       "The receiver's model has changed.  Redisplay the receiver."

       self displayObject
```

- #`anAspect` will be ignored, so we create a method `displayObject`, which is called by `displayOn:`. displayObject displays the on/off images for each lamp, depending on its state, then displays each image in its graphicsContext instance variable. graphicsContext is an instance of GraphicsContext, a feature-rich class used for drawing into a display surface, such as a view. More can be learned about this class from VisualWork's online documentation.

```
displayOn: ignored
       "Display the lamps in a window."

       self displayObject

displayObject
       "Display the lamps in the window."

       | image lamp |
       "model is a LampList"
       1 to: model size do:
              [ :index | lamp := model at: index.
                     lamp state = 0
                       ifTrue:
                         [ image := lamp getLampOffImage ]
                       ifFalse:
                         [ image := lamp getLampOnImage ].

              self graphicsContext displayImage: image
                 at: lamp position.].
```

- In `displayObject`, the methods `getLampOffImage` and `getLampOnImage` are used. These methods must be added to the Lamp instance methods. In addition to these methods, we need to add a class method, `initialize`, to create the images. You need not be concerned with the code within, just be aware that the `initialize` method exists.

```
getLampOffImage
       "Returns the image of the lamp in off state."

       ^LampOffImage

getLampOnImage
       "Returns the image of the lamp in on state."

       ^LampOnImage

initialize
```

```
"Initialize class with an image."

| bitPattern |

bitPattern := #[
                2r00001111 2r11110000
                2r00110000 2r00001100
                2r01000000 2r00000010
                2r10000000 2r00000001
                2r10000000 2r00000001
                2r10000000 2r00000001
                2r10000000 2r00000001
                2r10000000 2r00000001
                2r10000000 2r00000001
                2r01000000 2r00000010
                2r00100000 2r00000100
                2r00010000 2r00001000
                2r00001000 2r00010000
                2r00000100 2r00100000
                2r00000100 2r00100000
                2r00000010 2r01000000
                2r00000010 2r01000000
                2r00000010 2r01000000
                2r00000010 2r01000000
                2r00000010 2r01000000 ].

LampOnImage := Image
        extent: 16@20
        depth: 1
        palette: MappedPalette blackWhite
        bits: bitPattern
        pad: 8.

        LampOffImage := Image
        extent: 16@20
        depth: 1
        palette: MappedPalette whiteBlack
        bits: bitPattern
        pad: 8.
```

- `displayObject` still will not work properly. It references the position of each lamp, but until now the position has not been set. The position needs to be set somewhere, so we will set the position of each lamp in `LampList's` make method.

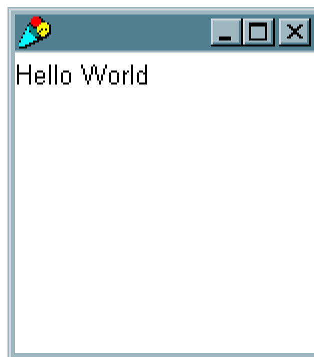
```
make: anInteger
        "Makes a lamp list with anInteger number of lamps input by
the user."

        | lamplist lamp |
        lamplist := LampList new: anInteger.
        anInteger
                timesRepeat:
                        [ lamp := Lamp new.
                          lamp position: 25 @ (lamplist size * 30).
                          lamp id: (lamplist size + 1).
                          lamp state: 0.
                          lamplist add: lamp.
                          lamp addDependent: lamplist].
        1 to: lamplist size do: [ :l |
                1 to: lamplist size do: [ :dep |
                        l = dep ifFalse: [
```

```
                                (lamplist at: l) addDependent: (lamplist
at: dep)]]].
       ^lamplist
```

- Now we have the methods to create a visual representation of a lamp, and the methods to update the view, but we still need to attach the model to the view and create the window to put the view in. To be displayed on the screen, a view must be contained in an instance of `ScheduledWindow`.

- Registering the view as a dependent of the model is simple through the use of the message `model: aModel`
  - `ScheduledWindow` has a model and a controller
- The “Scheduled” part of `ScheduledWindow` refers to the fact that `ScheduledWindow` is part of `ScheduledController`, the control manager.
  - Usually, the `ScheduledWindow` is created, and its visual components are added before it is opened. The following code demonstrates this:

```
| aWindow |
aWindow := ScheduledWindow new.
aWindow
       component: 'Hello World' asComposedText.
aWindow openIn: (20 @ 20 extent: 150 @ 150 ).
```



- Now we can create a new view, place it inside a window and register the model in the same class method for `LampView`:

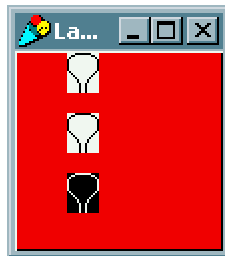
```
openOn: aLampList
       "Creates a new Lamp View on aLampList."

       | view window |
       view := self new.

       "Register the model"
       view model: aLampList.

       window := ScheduledWindow new.
       window label: 'Lamp Viewer'.
       window minimumSize: 50@100.
       window insideColor:
              (ColorValue red: 1.0 green: 0.0 blue: 0.0 ).
       window component: view.
       view window: window.
       window open.
       ^view.
```

- The last task that must be done is the simplest: create an instance of `LampView` in the `TrafficLight`. The simplest way is by modifying the `initialize` method:

```
initialize
    "Creates the three lights and turns the first on"

    lamplist := LampList make:3.
    lampView := LampView openOn: lamplist.
    ^self.
```

- The following code will create the light, initialize it to the second light, then changes lights every second for 10 seconds. The screen capture is the resulting window after 11 seconds (black was used as “on”).

```
| aTrafficLight |
aTrafficLight := TrafficLight new.
((aTrafficLight lamplist) at: 2 ) state: 1.
10 timesRepeat:
    [(Delay forSeconds: 1) wait.
    aTrafficLight changeLight].
```



Lecture 23: The Controller

- Now, suppose we want to use the code to create a control panel with Lamps, rather than a traffic light. We also want to control the lamps without entering commands into the workspace. We need to modify the controller.
  - Controllers serve two primary purposes, event handling and menu pop-ups. For now, we will focus on event handling.
    - Every controller has a `controlActivity` method which functions as an event handler. The `controlActivity` method is repeatedly invoked while control is active (e.g. the mouse pointer is in the view of a window). It is in this method that we check for events from the user, such as key presses and mouse button pushes, by sending messages to a sensor.
    - Each window has an input sensor, an instance of WindowSensor. The sensor holds queues for keyboard events and window sizing/moving/closing events. It also knows the state of the mouse, including the position of the pointer and the states of the buttons.
    - Lets start constructing the `controlActivity` method for the `LampController` by first checking for keyboard input. We'll use the number keys, 1, 2, and 3, to turn on the corresponding lamp. First we check to see if a key was pressed by using the `keyboardPressed` message:

```
sensor keyboardPressed.
```

      - If this message returns true, then a key has been pressed and we need to determine which one. The sensor will return the character pressed when we send it the `keyboard` message. We then convert the resulting character to an integer and test if it is a valid lamp number. If so, we set the state of the lamp to "on".
      - In the Smalltalk tradition, we want to keep the `controlActivity` method short, so we'll put the keyboard processing code in a separate method.

```
controlActivity
    "Do this when the mouse is in the window."

    (sensor keyboardPressed)
           ifTrue: [ self processKeyboard]

processKeyboard

    | int |
    int := sensor keyboard digitValue.
    (int between: 1 and: model size)
           ifTrue: [(model at: int) state: 1].
```

    - We want to add some way to quit the application, but request confirmation when the user chooses to quit. Lets implements this when the user presses the yellow (middle for 3 button mice, right for 2 button mice) mouse button. We can detect a mouse button by sending one of the following messages to the sensor:
      - `redButtonPressed`
      - `yellowButtonPressed`
      - `blueButtonPressed`.
    - In our case, we use `sensor yellowButtonPressed`. If this method returns true, then the `confirm:` message is sent to the Dialog class to bring up a window with "yes" and "no" buttons. Subsequent mouse presses are ignored until one of the confirm buttons is pressed. If the "yes" button is pressed, the confirm message returns true and the window is closed.

- Below we implement the yellow button activity method that is called when a yellow button press is detected in the `controlActivity` method.

```
controlActivity
       "Do this when the mouse is in the window."

       (sensor keyboardPressed)
             ifTrue: [ self processKeyboard]
             ifFalse: [
                    sensor yellowButtonPressed
                    ifTrue: [ self processYellowButton].
             ].

processYellowButton
       "This method is called when the yellow button
       is pressed"

       (Dialog confirm: 'Quit ?')
             ifTrue: [
               view window controller closeAndUnschedule].
```

- Lastly, it would be convenient if each lamp would turn on (and turn off all other lamps) by simply clicking on it. We will use the red (left mouse button) for this operation. As with the yellow button, we detect the red button press in the `controlActivity` method `sensor redButtonPressed`, then send the `processRedButton` message if the result is true.
- To determine if a lamp was clicked on, we compute a Rectangle (in view coordinates) which bounds the lamp image. Then we check to see if the point where the red button was clicked is contained in the bounding rectangle.
- We iterate through the LampList until we find a lamp that has been clicked on, in which case we change that lamp's state to #on, or until we have examined all lamps.
- The following code implements the algorithm discussed above:

```
processRedButton
   "This method is called when the red button is pressed."

   | mpt image box |

   "Wait for the mouse button to be released."
   sensor waitNoButton.

   "Get the point where the red mouse button was last
             pressed down."
   mpt := sensor lastDownPoint.

   "Assuming all lamp images are the same, get the
             first lamp image to use for computation"
   image := (model at: 1) getLampOffImage.

       "Now iterate through each lamp in the model or
             until we find one that has been clicked on."
   1 to: (model size) do: [ :lampNumber | | lamp |
          lamp := (model at: lampNumber).

             "Compute the bounding box of this lamp's image
                    in the view's coordinates."
          box := Rectangle origin: (lamp position)
                    extent: (image extent).

          "Check if the pointer was on the image when
                the button was pressed."
```

```
        (box containsPoint: mpt) ifTrue: [
              "If so, then turn that lamp on."
              ^lamp state: 1].
  ].
```

- Finally, included for completeness is the class definition as well as the complete `controlActivity` method.

```
Controller subclass: #LampController
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples-Lamps'

controlActivity
   "Do this when the mouse is in the window."

   (sensor keyboardPressed)
         ifTrue: [ self processKeyboard]
         ifFalse: [
               sensor yellowButtonPressed
               ifTrue: [ self processYellowButton].
               sensor redButtonPressed
               ifTrue: [ self processRedButton].
```