Objects

Some Background

Some basic features of object oriented programming include

- mutable state information
- messages and methods
- inheritance

Some people add other features, or place different emphasis on different features.

Origins of OOP

- Earliest example: *Simula*, a simulation language.
- First "pure" object oriented language: *Smalltalk*.
- Smalltalk was used for interacting with graphical objects, building graphical user interfaces.
- The Macintosh user interface inherits many features from the Smalltalk-based Xerox system.

Some Variations

- Languages/systems can be purely object oriented or just support object oriented programming.
- Two different approaches are used:
  - class/inheritance systems
  - prototype/delegation systems
- Most newer systems support some form of multiple inheritance.

Some OOP Systems

- *C++* from ATT
- *Objective C*, used in NeXT computers.
- *Smalltalk-80*
- *Object Pascal*, for programming the Macintosh.
- *Flavors*, on *Symbolics* Lisp machines
- *Lisp Object Oriented Programming System (LOOPS)*
- *Object Lisp*
- *Portable Common LOOPS*
- *CLOS*, the new Common Lisp standard
- *Common Objects*, from HP Labs.

References on OOP

Abelson, H. and Sussman, G. J., (1985), *Structure and Interpretation of Computer Programs*, Cambridge, Ma: MIT Press.

Winston, P. H. and Horn, B. K. P., (1988), *Lisp*, 3rd Edition, Reading, Ma: Addison Wesley.

Keene, S. E., (1989), *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Reading, Ma: Addison Wesley.

Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA), in *ACM SIGPLAN Notices*, (1986-91).

Wegner, P., (1990), "Concepts and paradigms of object-oriented programming," *ACM SIGPLAN Messenger*, vol. 1.

Introduction

Suppose we would like to build an “intelligent” function for describing or plotting a data set.

Initially we are looking at three types of data:

- single samples
- multiple samples
- paired samples

A `describe-data` function might be written as

```
(defun describe-data (data)
  (cond
   ((single-sample-p data) ...)
   ((multiple-sample-p data) ...)
   ((paired-sample-p data) ...)
   (t (error "don’t know how to
              describe this data set"))))
```

Now suppose we come up with a fourth type, a simple time series.

We would have to edit this function, possibly breaking good code.

An alternative is to arrange for the appropriate code to be looked up in the data set itself – to dispatch on the data set type.

We would need some way of defining new data types and actions.

This is the basis of object oriented programming.

Some Terms

An *object* is a data structure that contains information in *slots*.

Objects respond to *messages* asking them to take certain actions.

A message is *sent* to an object using the function **`send`** in an expression like

```
(send <object> <selector> <arg-1> ...  <arg-n>)
```

The <*selector*> is a keyword symbol used to identify the message.

As an example, the `histogram` function returns a histogram object:

```
> (setf p (histogram (normal-rand 20)))
#<Object: 1750540, prototype = HISTOGRAM-PROTO>
```

The object can be sent a message to add an additional set of points:

```
> (send p :add-points (normal-rand 20))
NIL
```

The *message selector* for the message is `:add-points`.

The *message argument* is the sample of normal variables generated by the expression `(normal-rand 20)`.

The *message* consists of the selector and the arguments.

The procedure called to respond to this message is the *method* for the message.

Inheritance and the Root Object

Objects are organized in an *inheritance hierarchy.*

At the top is the *root object*, the value of `*object*`.

The root object contains methods for standard messages, such as

`:own-slots` – list of slots in an object

`:add-slot` – insert a new slot

`:delete-slot` – delete a slot

`:slot-value` – access or modify a slot's value

Some examples:

```
> (send *object* :own-slots)
(DOCUMENTATION PROTO-NAME INSTANCE-SLOTS)

> (send *object* :slot-value 'proto-name)
*OBJECT*

> *object*
#<Object: 302253140, prototype = *OBJECT*>

> (send *object* :slot-value 'instance-slots)
NIL
```

Constructing New Objects

The root object is a *prototype* object.

Prototypes serve as templates for building new objects – *instances*.

The `:new` message is used to ask a prototype for a new instance.

Let's use an object to represent a data set:

```
> (setf x (send *object* :new))
#<Object: 1750672, prototype = *OBJECT*>
```

Initially, the object has no slots of its own:

```
> (send x :own-slots)
NIL
```

We can add a slot for some data and a slot for a title string:

```
> (send x :add-slot ’data (normal-rand 50))
(1.39981 -0.0601746 ....)
> (send x :add-slot ’title)
NIL
> (send x :own-slots)
(TITLE DATA)
> (send x :slot-value ’title)
NIL
> (send x :slot-value ’title "a data set")
"a data set"
> (send x :slot-value ’title)
"a data set"
```

Defining New Methods

The code used to respond to a message is called a *method*.

`send` searches the receiving object and its ancestors until it finds a method.

All messages used so far used methods in the root object.

New methods can be defined using **`defmeth`**.

The general form of a **`defmeth`** expression is:

```
(defmeth <object> <selector> <parameters>
  <body>)
```

The argument <*object*> is evaluated, the others are not.

An example:

```
(defmeth x :describe (&optional (stream t))
  (let ((title (slot-value 'title))
        (data (slot-value 'data)))
    (format stream "This is ~a~%" title)
    (format stream
            "The sample mean is ~a~%"
            (mean data))
    (format stream
            "The sample SD is ~a~%"
            (standard-deviation data))))
```

The definition looks like a `defun`, except for the object expression appearing as the first argument.

The new `:describe` method can be used like any other method:

```
> (send x :describe)
This is a data set
The sample mean is 0.127521
The sample SD is 1.0005
NIL
```

A few notes:

- Within a method, the variable `self` refers to the object receiving the message.
- `self` is needed since methods can be inherited.
- `self` is added to the environment for the function body before the other arguments.
- Within a method, the function `slot-value` can be used to access a slot.

  Using this function is more efficient than using the `:slot-value` message.
- The `slot-value` function can *only* be used in the body of a method.
- Within a method, `slot-value` can be used as a place form with `setf`.

It is good programming practice not to assume too much about the slots an object has.

For our data set object, we can write *accessor methods* for the title and data slots:

```
(defmeth x :title (&optional (title nil set))
  (if set (setf (slot-value 'title) title))
  (slot-value 'title))

(defmeth x :data (&optional (data nil set))
  (if set (setf (slot-value 'data) data))
  (slot-value 'data))
```

Using these accessors we can rewrite the `:describe` method:

```
(defmeth x :describe (&optional (stream t))
  (let ((title (send self :title))
        (data (send self :data)))
    (format stream "This is ~a~%" title)
    (format stream
            "The sample mean is ~a~%"
            (mean data))
    (format stream
            "The sample SD is ~a~%"
     (standard-deviation data))))
```

Printing Objects

The printing system prints objects by sending them the `:print` message.

A method for `:print` must take an optional *stream* argument.

A simple `:print` method for our data set:

```
(defmeth x :print (&optional (stream t))
  (format stream "#<~a>" (send self :title)))
```

The result is

```
> x
#<a data set>
```

Help Information for Messages

If we change the definition of the `:describe` method to

```
(defmeth x :describe (&optional (stream t))
"Method args: (&optional (stream t))
Prints a simple description of the object
to STREAM."
  (let ((title (send self :title))
        (data (send self :data)))
    ...))
```

then the string is used as a documentation string:

```
> (send x :help :describe)
:DESCRIBE
Method args: (&optional (stream t))
Prints a simple description of the object
to STREAM.
NIL
```

Prototypes

Creating Simple Prototypes

Instead of repeating the process used so far for each new data set, we can construct a *dataset prototype* using `defproto`.

The simplest form of a `defproto` expression is

(defproto <*name*> <*instance slots*>)

`defproto` evaluates <*instance slots*> and

- constructs a new object, assigns it to the global variable <*name*>
- installs a slot `proto-name` with value `name`
- installs a slot `instance-slots` with the symbols from <*instance slots*>
- installs a slot with value nil for each symbol in the `instance-slots` list

Different instances created from a prototype usually differ only in the values of their instance slots.

We can define a prototype for our data set as

```
(defproto data-set-proto '(data title))
```

Prototype objects are like any other objects.
We can place values in their slots

```
(send data-set-proto :slot-value
      'title "a data set")
```

and we can define methods for them:

```
(defmeth data-set-proto
         :title (&optional (title nil set))
  (if set (setf (slot-value 'title) title))
  (slot-value 'title))

(defmeth data-set-proto :data ...)

(defmeth data-set-proto :describe ...)

(defmeth data-set-proto :print ...)

(defmeth data-set-proto :plot ()
  (histogram (send self :data)
             :title (send self :title)))
```

Creating Instances from Prototypes

An instance is created by sending a prototype the `:new` message.

The method for this message does the following:

- creates a new object that inherits from the prototype
- adds a slot for each symbol in the prototype's `instance-slots` list
- sets the value of each new slot to its value in the prototype
- sends the new object the `:isnew` message with the arguments given to `:new`
- returns the new object

The root object `:isnew` initialization method allows slots to be initialized by keyword arguments:

```
> (setf x (send data-set-proto :new
               :data (chisq-rand 20 5)))
#<a data set>
> (send x :title)
"a data set"
> (send x :data)
(10.7254 10.4314 2.32346 ...)
> (send x :describe)
This is a data set
The sample mean is 5.17844
The sample SD is 3.2129
```

Any data set needs its own data.

We can define an `:isnew` method that requires a data argument:

```
(defmeth data-set-proto
        :isnew (data &key title)
  (send self :data data)
  (if title (send self :title title)))
```

It is often useful to define an appropriate `:isnew` initialization method.

It is (almost) never necessary to define a new `:new` method.

Prototypes and Inheritance

Prototypes can inherit from other objects, usually other prototypes.

The full form of a **defproto** expression is

```
(defproto <name>
          <instance slots>
          <shared slots>
          <parents>
          <doc string>)
```

All arguments except the first are evaluated.

<*instance slots*> and <*shared slots*> should be **nil** or lists of symbols.

We won't be using shared slots.

<*parents*> should be a single object or a list of objects – the default is the root object.

The instance slots of the new prototype are the union of the <*instance slots*> in the call and the instance slots of all ancestors.

So you only need to specify *additional* slots you want.

An equally-spaced time series prototype might add `origin` and `spacing` slots:

```
(defproto time-series-proto
          '(origin spacing) () data-set-proto)
```

Accessor methods for the new slots:

```
(defmeth time-series-proto
         :origin (&optional (origin nil set))
  (if set (setf (slot-value 'origin) origin))
  (slot-value 'origin))

(defmeth time-series-proto
         :spacing (&optional (sp nil set))
  (if set (setf (slot-value 'spacing) sp))
  (slot-value 'spacing))
```

and default values for the `title` slot and the new slots can be set as

```
(send time-series-proto :title "a time series")
(send time-series-proto :origin 0)
(send time-series-proto :spacing 1)
```

An example – a short moving average series:

```
> (let* ((e (normal-rand 21))
         (i (iseq 1 20))
         (d (+ (select e i)
               (* 0.6 (select e (- i 1))))))
    (setf y (send time-series-proto :new d)))
#<a time series>
> (send y :describe)
This is a time series
The sample mean is 0.194134
The sample SD is 1.15485
```

Overriding and Modifying Inherited Methods

The inherited `:plot` and `:describe` methods are not optimal for a time series.

We can *override* the `:plot` method with the definition

```
(defmeth time-series-proto :plot ()
  (let* ((data (send self :data))
         (start (send self :origin))
         (step (send self :spacing))
         (n (length data)))
    (plot-points (+ start (* step (iseq n)))
                 data)))
```

Instead of overriding the `:describe` method completely, we can augment it:

```
(defmeth time-series-proto
         :describe (&optional (stream t))
  (let ((ac (autocor (send self :data))))
    (call-next-method stream)
    (format stream
            "The autocorrelation is ~a~%"
            ac)))
```

The `call-next-method` function calls the next method for the current selector

(call-next-method *<arg-1>* ... *<arg-n>*)

The `autocor` function can be defined as

```
(defun autocor (x)
  (let ((n (length x))
        (x (- x (mean x))))
    (/ (mean (* (select x (iseq 0 (- n 2)))
                (select x (iseq 1 (- n 1)))))
       (mean (* x x)))))
```

Some Cautions

- Unfortunately the object system does not allow you to hide slots and messages intended for internal use from public access.
- To protect yourself against breaking inherited methods you need to follow a few guidelines:
  - Don't use slot names you didn't add yourself.
  - Don't modify slots unless you are sure it is safe.
  - Don't define methods for messages unless you know this won't do any harm.

Examples

Survival Function Estimator

We can use an object to organize our tools for fitting a survival function.

Since we need the unique death times, death counts, and numbers at risk, we can define a prototype that adds slots to hold these values:

```
(defproto survival-proto
          '(death-times num-deaths num-at-risk)
          ()
          data-set-proto)

(send survival-proto :title
      "a survival data set")
```

Accessors for these three slots are defined as

```
(defmeth survival-proto :death-times ()
  (slot-value 'death-times))
(defmeth survival-proto :num-deaths ()
  (slot-value 'num-deaths))
(defmeth survival-proto :num-at-risk ()
  (slot-value 'num-at-risk))
```

The **data** slot can be used to hold both the times and the death indicators.

The accessor method can be defined as

```
(defmeth survival-proto :data
         (&optional times status)
  (when times
    (call-next-method (list times status))
    (let* ((i (which (= 1 status)))
           (dt (select times i))
           (dt-list (coerce dt 'list))
           (udt (sort-data (remove-duplicates
                            dt-list
                            :test #'=)))
           (d (mapcar
               #'(lambda (x)
                 (count x dt-list :test #'=))
               udt))
           (r (mapcar
               #'(lambda (x)
                 (count x times :test #'<=))
               udt)))
      (setf (slot-value 'death-times) udt)
      (setf (slot-value 'num-deaths) d)
      (setf (slot-value 'num-at-risk) r)))
  (slot-value 'data))
```

The `:isnew` method fills in the data:

```
(defmeth survival-proto :isnew
         (times status &optional title)
  (send self :data times status)
  (if title (send self :title title)))
```

A `:describe` method:

```
(defmeth survival-proto :describe
         (&optional (stream t))
  (let ((title (send self :title))
        (data (send self :data)))
    (format stream "This is ~a~%" title)
    (format stream
            "The mean time is ~a~%"
            (mean (first data)))
    (format stream
            "The number of failures is ~a~%"
            (sum (second data)))))
```

A `:plot` method:

```
(defmeth survival-proto :plot ()
  (let ((km (send self :km-estimator))
        (udt (send self :death-times)))
    (plot-lines (make-steps udt km))))
```

A method for the Kaplan-Meier estimator:

```
(defmeth survival-proto :km-estimator ()
  (let ((r (send self :num-at-risk))
        (d (send self :num-deaths)))
    (accumulate #'* (/ (- r d) r))))
```

Methods for the Fleming-Harrington estimator and standard errors based on Greenwood's formula and Tsiatis' formula are

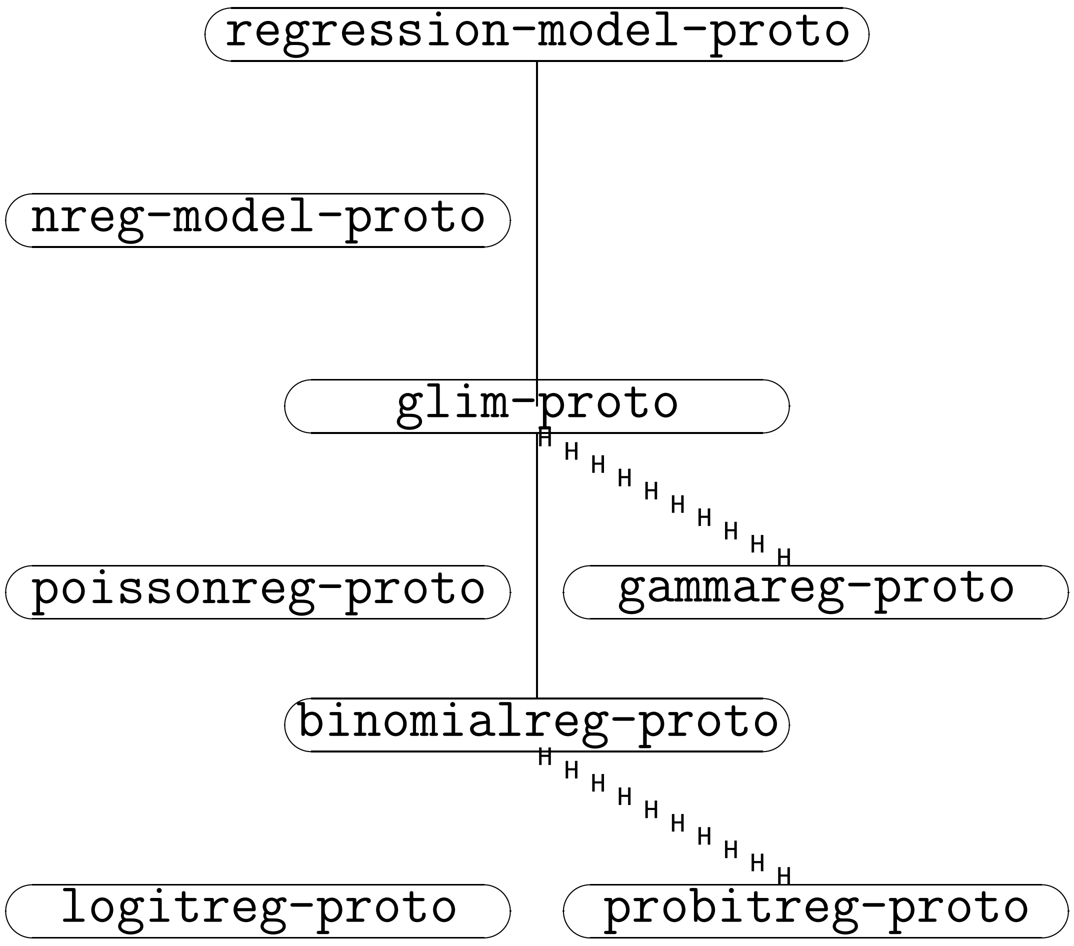
```
(defmeth survival-proto :fh-estimator ()
  (let ((r (send self :num-at-risk))
        (d (send self :num-deaths)))
    (exp (- (cumsum (/ d r))))))

(defmeth survival-proto :greenwood-se ()
  (let* ((r (send self :num-at-risk))
         (d (send self :num-deaths))
         (km (send self :km-estimator))
         (rd1 (pmax (- r d) 1)))
    (* km (sqrt (cumsum (/ d r rd1))))))

(defmeth survival-proto :tsiatis-se ()
  (let ((r (send self :num-at-risk))
        (d (send self :num-deaths))
        (km (send self :km-estimator)))
    (* km (sqrt (cumsum (/ d (^ r 2)))))))
```

Model Prototypes

Linear regression, nonlinear regression and GLIM model prototypes are arranged as:



OOP and User Interfaces

Users of a graphics workstation can take a limited set of actions.

They can

- hit keys
- move the pointer
- click the pointer

These actions can be viewed as messages that are being sent to objects on the screen.

Graphical user interfaces present users with a model based on a desk:

- Overlapping windows act like sheets of paper
- Different windows represent different software objects on the screen
- Each object may respond differently to key stroke or locator click messages

The *user interface conventions* of the window system determine what objects receive what messages.

Several variations are possible:

- On the Macintosh the front window receives all input.

  A window becomes the front window by clicking on it.
- Under *twm* in an *X11* system, the window containing the locator cursor receives all input.

Most window management systems are object-oriented in design.

The window system is responsible for converting user actions into messages.

The programmer is responsible for defining any methods objects need in order to respond to the user's actions.

Some other variations include:

- size and location of windows may have to be determined by the user
- resizing or moving windows from within a program may not be possible
- menus may be popped up or pulled down
- mouse buttons may mean different things in different settings
- mice have different numbers of buttons

Menus

Menus are created using the **`menu-proto`** prototype.

The initialization method requires a single argument, the menu title.

```
> (setf data-menu
        (send menu-proto :new "Data"))
#<Object: 4055334, prototype = MENU-PROTO>
```

A menu bar is available for holding menus not associated with a graph.

The **`:install`** message installs a menu; **`:remove`** removes it:

```
> (send data-menu :install)
NIL
> (send data-menu :remove)
NIL
```

Initially a menu contains no items.

Items are created using the `menu-item-proto` prototype.

The initialization method requires one argument, the item's title.

It also accepts several keyword arguments, including

- `:action` – a function to be called when the item is selected
- `:enabled` – true by default
- `:key` – a character to serve as the keyboard equivalent
- `:mark` – `nil` (the default) or `t` to indicate a check mark.

Analogous messages are available for accessing and changing these values in existing menu items.

Suppose we assign a data set object as the value of some global symbol:

```
(setf *current-data-set* x)
```

Two menu items for dealing with the current data set are:

```
> (setf desc
        (send menu-item-proto :new "Describe"
              :action
              #'(lambda ()
                (send *current-data-set*
                      :describe))))
#<Object: 4034406, prototype = MENU-ITEM-PROTO>
> (setf plot
        (send menu-item-proto :new "Plot"
              :action
              #'(lambda ()
                (send *current-data-set*
                      :plot))))
#<Object: 3868686, prototype = MENU-ITEM-PROTO>
```

You can force an item's action to be invoked by sending it the `:do-action` message:

```
> (send desc :do-action)
This is a data set
The sample mean is 4.165463
The sample SD is 1.921366
```

The system sends the `:do-action` message when you select the item in a menu.

The `:append-items` message adds the items to the menu:

```
> (send data-menu :append-items desc plot)
NIL
```

On the Macintosh, the `:key` message adds a keyboard equivalent to an item:

```
> (send desc :key #\D)
#\D
```

This message is ignored on most other systems.

You can also enable and disable an item with the `:enabled` message:

```
> (send desc :enabled)
T
> (send desc :enabled nil)
NIL
> (send desc :enabled t)
T
```

Before a menu is presented, the system sends each of its menu items the `:update` message.

You can define a method for this message that enables or disables the item, or adds a check mark, as appropriate.

Dialogs

Dialogs are similar to menus in that they are based on a dialog prototype and dialog item prototypes.

There are, however many more variations.

Fortunately most dialogs you need fall into one of several categories and can be produced by custom dialog construction functions.

Modal Dialogs

Modal dialogs are designed to ask specific questions and wait until they receive a response.

All other interaction is disabled until the dialog is dismissed – they place the system in dialog mode.

Six functions are available for producing some standard modal dialogs:

- `(message-dialog` <*string*>`)` – presents a message with an **OK** button; returns `nil` when the button is pressed.
- `(ok-or-cancel-dialog` <*string*>`)` – presents a message with an **OK** and a **Cancel** button; returns `t` or `nil` according to the button pressed.

- `(choose-item-dialog` *<string> <string-list>*`)` – presents a heading and a set of radio buttons for choosing one of the strings.

  Returns the index of the selected string on **OK** or `nil` on **Cancel**.

  ```
  > (choose-item-dialog
     "Dependent variable:"
     '("X" "Y" "Z"))
  1
  ```

- `(choose-subset-dialog` *<string> <string-list>*`)` – presents a heading and a set of check boxes for indicating which items to select.

  Returns a list of the list of selected indices on **OK** or `nil` on **Cancel**.

  ```
  > (choose-subset-dialog
     "Independent variables:"
     '("X" "Y" "Z"))
  ((0 2))
  ```

- `(get-string-dialog` <*prompt*> `[:initial` <*expr*>`])`
  presents a dialog with a prompt, an editable text field, an **OK** and a **Cancel** button.
  The initial contents of the editable field is empty or the `~a`-formated version of <*expr*>.
  The result is a string or `nil`.

```
> (get-string-dialog
   "New variable label:"
   :initial "X")
"Tensile Strength"
```

- `(get-value-dialog` <*prompt*> `[:initial` <*expr*>`])`
  like `get-string-dialog`, except
  - the initial value expression is converted to a string with `~s` formatting
  - the text is interpreted as a lisp expression and is evaluated
  - the result is a list of the value, or `nil`

There are two dialogs for dealing with files:

- `(open-file-dialog)` – presents a standard **Open File** dialog and returns a file name string or `nil`.

  Resets the working folder on **OK**.
- `(set-file-dialog prompt)` – presents a standard **Save File** dialog.

  Returns a file name string or `nil`.

  Resets the working folder on **OK**.

In the *X11* versions of these are currently just `get-string-dialogs`.

The MS Windows versions are also not yet fully developed.

Modeless Dialogs

Two standard modeless slider dialogs are available:

- `sequence-slider-dialog` – a slider for scrolling through a sequence.

  An action function is called each time the slider is adjusted.

  The current sequence value or a value from a display sequence is displayed.

- `interval-slider-dialog` – similar to the sequence slider, except that a range to be divided up into a reasonable number of points is given.

  By default, the range and number of points are adjusted to produce nice printed values.

Custom Dialogs

If the standard dialogs are not sufficient, you can construct custom modal or modeless dialogs using a variety of items:

- static and editable text fields
- radio button groups
- check boxes
- sliders
- scrollable lists
- push buttons