GeneralClasses

COLLABORATORS

	TITLE : GeneralClasses		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 26, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

GeneralClasses

1.1 Descriptions of the Methods of the General classes:

WARNING: The documentation in this file is from the Original Little
SmallTalk documentation. If there is any question of whether
these documents are correct, you should check the corresponding
source file in AmigaTalk:General/ directory in order
to determine what is currently implemented.

Undocumented Classes or methods are probably NOT something that the
casual User should be concerned with. Some methods & Classes are
purposely NOT documented.

Show below is the hierarchy of the General Classes that are loaded
into memory before the AmigaTalk system is ready for user input.

The indentations indicate which classes are sub-classes:

Object

UndefinedObject

Symbol

DependencyTransformer -- Added for V2.5+

Model -- Added for V2.5+

ValueModel -- Added for V2.5+

ComputedValue -- Added for V2.5+

BlockValue -- Added for V2.5+

PluggableAdaptor -- Added for V2.5+

ProtocolAdaptor -- Added for V2.5+

AspectAdaptor -- Added for V2.5+

IndexedAdaptor -- Added for V2.5+

SlotAdaptor -- Added for V2.5+

ValueHolder -- Added for V2.5+

1.2 Object Class:

The class Object is a superclass of all classes in the system, and is

used to provide a consistent basic functionality and default behavior.

Many methods in class Object are overridden in subclasses.

NOTE: Unlike most other versions of Smalltalk, AmigaTalk does not have

dependencies for every object, only the Model class & its subclasses

use them.

Examples: Printed result:

7 ~~ 7.0 True

7 asSymbol #7

7 class Integer

7 copy 7

7 isKindOf: Number True

7 isMemberOf: Number False

7 isNil False

7 respondsTo: #+ True

The methods for Object Class are:

instVarAt: index

Answer with a fixed variable in an object. The numbering

of the variables corresponds to the named instance variables.

Fail if the index is not an Integer or is not the index

of a fixed variable. The range for index is 1 to Object size.

This method is currently only used by the SlotAdaptor class.

instVarAt: anInteger put: anObject

Store a value into a fixed variable in the receiver.

The numbering of the variables corresponds to the named

instance variables. Fail if the index is not an Integer or

is not the index of a fixed variable.

Answer with the value stored as the result. The range for

index is 1 to Object size. Using this message violates

the principle that each object has sovereign control over

the storing of values into its instance variables. This

method is currently only used by the SlotAdaptor class.

identityHash

Return an Integer that identifies (hashes) to the object.

== or =

Return true if receiver and argument are the same object, false

otherwise.

~~ or ~=

Inverse of ==.

asString

Return a string representation of the receiver, by default this is the
same as printString, although one or the other is redefined
in many subclasses.

asSymbol

Return a symbol representing the receiver.

yourself

This is just a synonym for self.

class

Return object representing the class of the receiver.

copy

Return shallowCopy of receiver. Many subclasses redefine shallowCopy.

deepCopy

Return the receiver. This method is redefined in many subclasses.

first

Return first item in sequence, which is by default simply the receiver.
See next, below.

do: aBlock

The argument must be a one argument block. Execute the block on every
element of the receiver collection. Elements in the receiver collection are listed using first and next, so the default
behavior is merely to execute the block using the receiver as argument.

do: aBlock without: anObject

The first argument must be a one argument block. Execute the block on every
element of the receiver collection except for anObject. Elements in the receiver collection are listed using first and next, so the default
behavior is merely to execute the block using the receiver as argument.

error: errMsg

Argument must be a String. Print argument string as error message.
Return nil.

isKindOf: className

Argument must be a Class. Return true if class of receiver, or any
superclass thereof, is the same as argument.

isMemberOf: className

Argument must be a Class. Return true if receiver is instance of
argument class.

ifKindOf: className thenDo: aBlock

If the class of the receiver, or any superclass thereof, is the same
as the argument, then execute aBlock.

isNil

Test whether receiver is object nil.

next

Return next item in sequence, which is by default nil. This
message is redefined in classes which represent sequences, such as
Array or Dictionary.

notNil

Test if receiver is not object nil.

print

Display print image of receiver on the Status Window.

printString

Return a string representation of receiver. Objects which do not redefine printString, and which therefore do not have a printable
representation, return their class name as a string.

respondsTo: msgSymbol

Argument must be a symbol. Return true if receiver will respond to
the indicated message.

shallowCopy

Return the receiver. This method is redefined in many subclasses.

asciiToString: aNumber

Convert aNumber into a single-character String .

subclassResponsibility: methodString

Inform the user that a subclass did NOT implement the given method.

notImplemented: methodString

Inform the user that the given method is NOT implemented.

doesNotUnderstand: methodString

Inform the user that a subclass does NOT understand the given method.

shouldNotImplement: methodString

Inform the user that a subclass should NOT implement the given method.

notYetImplemented

Inform the user that a method is NOT implemented yet.

breakPoint: msgString

This method will act as a breakpoint for your code, displaying
a Requester with msgString in it. This does not mean that
your code will stop executing, this method is simply a way of
displaying debugging statements. In the future, perhaps it will do
more than simply display a message on the GUI.

The following methods were added to support inter-object communications:

perform: selector

Send the unary selector (Symbol)to the receiver.

perform: selector orSendTo: otherTarget

If I wish to intercept and handle selector myself,

do it; else send it to otherTarget. Default behavior

is to execute ˆ otherTarget perform: selector.

perform: selector with: anObject

Send the selector (Symbol), to the receiver with

anObject as its argument.

perform: selector withArguments: argArray

Send the selector, (Symbol), to the receiver with

arguments in argArray. Fail if the number of

arguments expected by the selector does not match

the size of argArray. This method is the general case

of the perform: methods. Use perform:with: for a single argument

method, perform:with:with: for methods that require two arguments,

and perfrom:with:with:with: for methods that require three arguments.

perform: is for methods that require no arguments.

perform: selector with: firstObject with: secondObject

Send the selector, (Symbol), to the receiver with the

given arguments.

perform: selector with: firstObject with: secondObject with: thirdObject

Send the selector, (Symbol), to the receiver with the given arguments.

performUpdate: aSymbol with: anObject

This method is a synonym for perform:with:.

performUpdate: aSymbol

This method is a synonym for perform:.

1.3 Model Class:

Added for V2.5+

Any Model can have dependents that receive notification of

any change to the object. This representation is faster but

takes more space.

Instance Variables:

myDependents <nil | Object | DependentsCollection>

haveAChange <Integer>

linkedMethods <KeyedCollection>

Model is an abstract class whose subclasses represent various
kinds of information models. An information model is an
object on which user-interface objects such as input fields
depend for their data -- thus, the interface objects are
said to be dependents of the model.
Here, dependents are kept in an instance variable.
While Model does not provide any new abilities, it has many
subclasses that do. An ApplicationModel mediates between a
set of data models and the user interface that is used to
manipulate the data. Various kinds of ValueModel are able to
adapt simple data objects so they behave like full-fledged models.
Available methods are:
postCopy
Do not copy the dependents list, just everything else.
initialize
addDependent: anObject
Make the given object one of the receiver's dependents.
changeComplete
If haveAChange count has reached zero, return true, else
return false.
changeMade
Increment the haveAChange instance variable.
release
Remove all of the receiver's dependents.
breakDependents
Remove all of the receiver's dependents.
canDiscardEdits
Answer true if none of the controllers on this model has
unaccepted edits that matter.
dependents: dependentsOrNil
Set the receivers dependents.
dependents
Answer a collection of objects that are 'dependent' on the
receiver; that is, all objects that should be notified if
the receiver changes.
hasUnacceptedEdits
Answer true if any of the controllers on this model has unaccepted edits.
removeDependent: anObject
Remove the given object as one of the receiver's dependents.

topController

Find the first top controller on me. Is there any danger

of their being two with the same model? Any danger

from ungarbage collected old controllers?

linkMethod: classAndMethod

Make a message list and put this method in it.

value

SubClasses must override this.

perform: selector orSendTo: otherTarget

Selector was just chosen from a Control by a User.

If I can respond, then perform it on myself. If not, send

it to otherTarget.

See Also, ValueModel

1.4 ValueModel Class:

Added for V2.5+

ValueModel Class is a simple Model that provides direct access

to some kind of value. It notifies dependents when the value changes.

The collection accessing protocol is here as a convenience, to

avoid some of the need for special collection models.

Subclasses must implement the following messages:

value

setValue:

ValueModel is an abstract class that provides model-like abilities for

an enfolded object -- that is, it notifies dependent objects whenever

its held object is changed.

Value models are most commonly used to hold data models on which gadgets

such as input fields depend. To understand why a value model is needed,

take the case of an input field that displays a number. The input

field needs to be notified whenever the application changes the number,

so it registers itself as a dependent of the number. If that dependency

were established on the raw number, however, the dependency would be

obsolete as soon as the application substituted a different number,

defeating its purpose. Instead, a value model is used to hold the

number, and the input field registers itself as a dependent of the

value model. The application can insert a new number and the value

model then notifies the input field of the change so the field can

get the new number and display it. Since the value model remains in place while its value is changed, the dependency that was established by the input field remains alive as long as the application is running. The most commonly used subclass of ValueModel is ValueHolder , which would be used in the simple case described above. Because value holders are widely used, every object inherits from the Object class the ability to enfold itself in a value holder -- sending #asValue to any object returns a ValueHolder on the object. A consequence of inserting a value holder as an adaptor or mediator between a data object and its dependents is that you must send #value to the value model, thus extracting the enfolded object, before you can send a message to that object. For example, suppose the numeric input field mentioned above uses an instance variable named salesCommission to hold its ValueHolder. If the application wanted to retrieve the actual number for use in a computation, it would use the expression 'self salesCommission value' instead of simply 'self salesCommission.' The fact that a value model always gets its enfolded object in response to #value, and sets that value in response to #value:, simplifies communications for gadgets.

Another commonly used subclass of ValueModel is AspectAdaptor , which is used to enfold an embedded value. For example, suppose we have an AccountNumber class that has a string part and a number part, for a composite account number such as 'TEL-4792'. We might want to use a separate input field for each part of this account number.

One AspectAdaptor could be used to enfold the string part, and another the numeric part. The most flexible subclass of ValueModel is PluggableAdaptor , because it can be configured to transform the value on its way to and from the dependent.

PluggableAdaptor can be configured with blocks to perform highly specialized transformations. Other subclasses of ValueModel are more specialized. A @" BlockValue " LINK "BlockValueClass"} enables a computation inside a block to have dependents.

An IndexedAdaptor enfolds an element in a collection.

SlotAdaptor and DependencyTransformer are mainly used by system machinery.

A ValueModel provides a convenient way for an application to arrange to receive a particular message whenever the value is changed. Making such an arrangement is known as expressing an interest in the value, and involves sending #onChangeSend:to:. Retracting an interest is achieved via #retractInterestsFor:. When creating a subclass, equip it with the following methods:

value

setValue:

The available methods are:

new

Create a new instance of ValueModel.

initialize

Initialize the instance. Subclasses may extend this.

release

Break the dependency links from any parts of myself to myself.

Subclasses holding composite values will implement this in

a non-trivial way.

releaseParts

Break the dependency links from any parts of myself to myself.

Subclasses holding composite values will implement this in

a non-trivial way.

setValue: newValue

Set the currently stored value, without notifying dependents. "

value

Answer the currently stored value.

value: newValue

Set the currently stored value, and notify dependents.

valueUsingSubject: aSubject

Return the value of aSubject.

asValue

Since the receiver is already a ValueModel, merely return self.

onChangeSend: aSymbol to: anObject

Arrange for anObject to receive a message named aSymbol when

I signal that my attribute #value has changed.

retractInterestsFor: anObject

Undo a send of onChangeSend:to:

compute: aBlock

Answer a BlockValue that computes aBlock with the receiver's value

as the argument. aBlock will become a dependent of the receiver,

and will be sent the message value: when the receiver is sent the

message value:.

receive: aSelector

Answer a BlockValue that responds to the message value by sending

aSelector as a message to the receiver. This BlockValue will become a

dependent of the receiver, and will be sent the message value: when the receiver is sent the message value:.

receive: aSelector with: value1

Answer a BlockValue that responds to the message value by sending aSelector as a message to the receiver. This BlockValue will become a dependent of the receiver, and will be sent the message value: when the receiver is sent the message value:. The message aSelector has one argument, value1. It is assumed that value1 itself responds to the message value (i.e., may be a kind of ValueModel).

with: value2 compute: aBlock

Answer a BlockValue that computes aBlock with the receiver and value2 as the first and second arguments, respectively. This BlockValue will become a dependent of the receiver, and will be sent the message value: when the receiver is sent the message value:. It is assumed that value2 itself responds to the message value (i.e., may be a kind of ValueModel).

with: value2 with: value3 compute: aBlock

Answer a BlockValue that computes aBlock with the receiver, value2, and value3 as the first, second and third arguments, respectively. This BlockValue will become a dependent of the receiver, and will be sent the message value: when the receiver is sent the message value:. It is assumed that the objects value2 and value3 respond to the message value (i.e., may be a kind of ValueModel).

isBuffering

ValueModels by default do not buffer values, only special subclasses who should reimplement this message for themselves.

See Also, BlockValue ,

ValueHolder

1.5 DependencyTransformer Class:

Added for V2.5+

DependencyTransformer transforms update messages from an object into concrete messages to a receiver.

Objects understand:

expressInterestIn: anAspect for: anObject sendBack: aSelector

by creating a DependencyTransformer that looks for upates on anAspect and sending aSelector to anObject

retractInterestIn: anAspect for: anObject

removes the DependencyTransformer created above.

ValueModels understand

onChangeSend: aSelector to: anObject retractInterestsFor: anObject

since ValueModels only do changed: #value

the aspect can be dropped.

Instance Variables:

receiver <Object> the object to receive a message on update

selector <Object> the message selector to send

numArguments <Object> number of arguments in the message selector

aspect <Object> the change aspect to look for

Available methods are:

setReceiver: aReceiver aspect: anAspect selector: aSymbol

aspect

Return the aspect.

receiver

Return the receiver.

selector

Return the selector.

matches: anObject forAspect: anAspect

Return true if anObject matches the receiver and the aspect.

update: anAspect with: parameters from: anObject

Update anObject.

= anObject

Two DependencyTransformers are equivalent if their receiver

aspect and selectors are identical.

hash

Redefined because = is redefined.

See Also, DependentsCollection

1.6 DependentsCollection Class:

Added for V2.5+

A DependentsCollection is a collection of dependents for some

object. Instances forward update messages to the dependents,

which are the elements of the collection. Note that the same

dependent may appear more than once in the collection. Note

also that the size of a DependentsCollection must always be

2 or greater. (If an object has only one dependent, that

object by itself serves as the collection of dependents.)

Available methods are:

asDependentsAsCollection

Answer the receiver, considered as a collection of

dependents, as a real Collection. Since the receiver

is a Collection already, answer the receiver.

asDependentsWith: anObject

Answer the receiver, considered as a collection of

dependents, with anObject added.

asDependentsWithout: anObject

Answer the receiver, considered as a collection of

dependents, with the first occurrence of anObject

(if any) removed. If anObject does not occur in

the receiver, answer the receiver.

If there is only one dependent left, just answer it,

rather than a new Collection.

performUpdate: aSymbol

Send aSymbol to each member of the receiver.

performUpdate: aSymbol with: anObject

Send aSymbol to each member of the receiver with

anObject as argument.

update: anAspect with: aParameter from: anObject

Send the message update: anAspect with: aParameter

from: anObject to each member of the receiver.

updateRequest

Send the message updateRequest to each member of the receiver.

If any member answers false, answer false; otherwise, answer true.

updateRequest: anAspectSymbol

Send the message updateRequest: to each member of the

receiver with anAspectSymbol as argument. If one

answers false, answer false, otherwise answer true.

See Also, DependencyTransformer

1.7 ComputedValue Class:

Added for V2.5+

ComputedValue is an abstract class that represents a computation

that propagates changes to dependents. It caches the result of the

computation. Kinds of ComputedValues can not respond to the message

value:. Subclasses must implement:

parts computeValue

Instance Variables:

cachedValue <Object>

eagerEvaluation <Boolean> controls whether to wait until the

receiver is asked to perform the

computation (eager vs. late)

unassignedValue <Object> a unique object that is guaranteed to

be used by no one else in the system,

used to denote that the value has not yet been computedObject

ComputedValue is an abstract class that provides support

for creating a BlockClosure that recomputes a cached value

whenever one of the block arguments changes its value. When asked

for its #value, a ComputedValue supplies its cached value,

recomputing it if necessary. This is useful when object3 is

computed using object1 and object2, which are expected to be

value models. ComputedValue has a single subclass,

BlockValue. For usage instructions, see BlockValue . When creating

a subclass, equip it with the following methods:

parts computeValue

Subclasses should not implement: value:

Available Class methods are:

initialize

Setup the instance variables.

unassigned

Return the contents of unassignedValue

releaseParts

Remove any dependencies involving the receiver.

eagerEvaluation: aBoolean

If aBoolean is true the receiver will do late evaluation of its

computation; otherwise the receiver will do eager computation.

parts

Answer a collection of objects that have the receiver as a dependent.

Defined in BlockValue Class (& any other subclasses).

value

Answer the cached value for the receiver. If the value is unknown,

then compute the value.

value: anObject

This method only reports that you should Not Implement it.

update: aspect with: parameter from: sender

If a model is propogating a change, then reset the receiver and propogate change to dependents.

printOn: aStream

Print the receiver's value on aStream.

computeValue

Compute a value for the receiver.

resetValue

Set the receiver's value to unknown. Propogate change to dependents.

See Also, BlockValue

1.8 BlockValue Class:

Added for V2.5+

An instance of BlockValue represents a computation that propagates changes. The value of an instance is recomputed by evaluating a block when the value of any of the instance's arguments changes. All dependents are notified when the value has changed.

Instance Variables:

block <BlockClosure>

arguments <SequenceableCollection of: ValueModel>

numArgs <SmallInteger> The number of arguments block takes

A BlockValue computes its value using a given BlockClosure and a set of arguments for that block. It registers itself as a dependent of each argument, so the arguments are typically value models. Whenever any of the argument objects changes its value, the BlockValue is updated. The result is similar to sending #onChangeSend:to: to each of the argument objects, asking them to trigger a method that updates a particular value. Using a BlockValue eliminates the need to create the method that is triggered -- instead, the block is evaluated with the changed arguments. The resulting value is cached so that it need not be recomputed unless one of the block arguments changes again. Being able to use a block instead of a method is especially helpful in the case of dialog that has been built from scratch, such that a method in the application model would have difficulty accessing some of the data in the dialog for its computation. A BlockValue is typically created by sending #block:arguments: to this class. The ValueModel class also provides a set of methods for conveniently spawning a BlockValue from one of the argument objects (see the constructing protocol in ValueModel).

By default, a BlockValue recomputes its cached value whenever it is

notified that one of the block arguments has changed. This is known
as eager evaluation. In some situations, late evaluation may be
preferable -- then, the cached value is only recomputed when it is
requested via #value (assuming one of the block arguments has changed).
Late evaluation can be arranged by sending an #eagerEvaluation:
message to the BlockValue with false as the argument. This is most
useful when the cached value is only requested infrequently and the
block computation is costly in terms of time or other resources.
Available methods are:
block: aBlock arguments: aCollection
Answer an instance of the receiver with aCollection as arguments.
with: aBlock
Answer a new instance of the receiver that computes aBlock.
dependOn: anObject
Make the receiver depend on anObject.
parts
Answer a collection of objects that have the receiver as a dependent.
computeValue
Compute a value for the receiver.
setBlock: aBlock
Set the block for the receiver to be aBlock.
setBlock: aBlock arguments: aCollection
Set the receiver's block to be aBlock and the arguments to
be aCollection.

1.9 PluggableAdaptor Class:

Added for V2.5+
PluggableAdaptors provide a level of indirection between a
Controller and an underlying model. The Controller sends my
instances the standard messages value & value:, which I
convert into arbitrary actions defined by blocks.
Instance Variables:
model <ValueModel> the underlying model (only used for
dependency and for isActive testing)
getBlock <BlockClosure> evaluate this block to get the value
putBlock <BlockClosure> evaluate this block to set the value
updateBlock <BlockClosure> evaluate this block to handle
an update from the model; if it

returns true, notify our dependents

The getBlock is evaluated with one argument, the model.

The putBlock is evaluated with two arguments, the model and the new value. The updateBlock is evaluated with three arguments, the model, the update aspect, and the update parameter.

We use blocks rather than selectors because blocks are much more flexible than selectors for representing encapsulated behavior. They can reference more than one object, and they can include embedded parameters such as a collection index.

A PluggableAdaptor is the most flexible of the value models, because its activities are highly configurable. This flexibility comes at the cost of a certain conceptual complexity, however.

At one time, PluggableAdaptor was the only value model -- now, more convenient value models exist for the most common situations in which a PluggableAdaptor was formerly applied.

A PluggableAdaptor has a model, which can either be an application model or a domain model, from which it obtains the desired data value. The adaptor is configured via three blocks, which enable it to perform customized actions at three junctures in the flow of communications between the dependent (typically a widget) and the model. The first block, the getBlock, controls what happens when a value is requested (via #value). The block takes one argument, the model. The block returns the value, after fetching it from the model and applying any necessary computations or transformations.

For example, the following getBlock fetches an accountNumber from the model, converts it to a string and pads it with leading zeroes:

```
[ :model | | paddedString |
paddedString <- model accountNumber printString.
(6 - paddedString size)
timesRepeat: [paddedString := ’0’, paddedString].
paddedString ]
```

The second block, the putBlock, controls what happens when a value is stored (via #value:). The block takes two arguments, the model and the value to be stored. The block stores the value in the model after applying any necessary computations or transformations.

For example, the following putBlock converts a padded accountNumber string back into a number and stores the number in the model:

```
[ :model :val | model accountNumber: val asNumber].
```

The third block, the updateBlock, controls what happens when the adaptor receives an #update:with:from: message. It receives that message whenever the model sends a variant of #changed:with: to itself -- in the accountNumber example, the model would send such a message when its accountNumber had been changed. The block takes three arguments: the model and the first two arguments from the #update:with: message (known as the update aspect and the update parameter). The block returns true or false, usually after testing the aspect to see whether the adaptor cares about that type of change in the model. When the updateBlock returns true, the adaptor's getBlock is invoked to update the widget's value. When the updateBlock returns false, no action is taken. For example, the following updateBlock causes the widget to refetch the value only when the update aspect is #accountNumber and the parameter (an accountNumber string) is less than 1000:

[:model :aspect :parameter |

aspect == #accountNumber and: [parameter asNumber < 1000]].

A PluggableAdaptor is created by sending #on: to this class, with the model as the argument. The three blocks are then initialized via #getBlock:putBlock:updateBlock.

Available methods are:

on: aModel

Create a new PluggableAdaptor with the given aModel Model

getBlock: aBlock1 putBlock: aBlock2 updateBlock: aBlock3

Set the blocks used for dealing with the model.

initialize

Initialize the blocks on the assumption that the underlying model is a ValueModel . This is wrong, of course.

model: aModel

Set our model instance variable to aModel.

subjectChannel: aValueHolder

collectionIndex: index

Initialize the receiver to access the given element of a collection that is the value of the model.

getSelector: aSymbol0 putSelector: aSymbol1

Initialize the receiver to act like the old pluggable classes.

performAction: aSelector

Initialize the receiver to perform the action when assigned a value selectValue: aValue

Initialize the receiver to act like a Boolean

that is true when the model’s value is equal to aValue.

model

Return our model.

setValue: newValue

value

Return our value.

valueUsingSubject: aSubject

update: aspect with: parameter from: sender

addDependent: aDependent

removeDependent: aDependent

isProtocolAdaptor

Answer as to whether the receiver transduces protocol

into ValueModel protocol.

makeAdaptorForRenderingStoreLeafInto: pair

renderingValueUsingSubject: aSubject

1.10 ProtocolAdaptor Class:

Added for V2.5+

Class ProtocolAdaptor is an abstract class that introduces the

concept of a ValueModel which redirects the value & value:

messages to another object (the subject) and adds lazy dependency.

Lazy dependency means that the ProtocolAdaptor will only register

itself as a dependent of the subject if it has at least one dependent

and the subject will send update messages. Subclasses which implement

value: are responsible for sending update messages if the subject

does not send update messages.

ProtocolAdaptors can have a collection of object used to transform

the subject into the "target" which subclasses then operate on.

A ProtocolAdaptor has either a constant or variable subject. The

constant subject is initialized with the subject: or

subject:sendsUpdates: messages. A variable subject is commonly used

when a set of adaptors all adapt a different part of the same subject

or the subject is changed after initialization. A variable subject

requires the use of a ValueModel to inform the ProtocolAdaptor of the

new subject and is initialized using the subjectChannel: or

subjectChannel:sendsUpdates: messages.

Dependents are notified of the value changing when the subject is changed.

Instance Variables:

subject <Object> The object we're adapting.

subjectSendsUpdates <Boolean> When this is set to true, it is

assumed that the subject will

send update notices and we'll pass them on to our dependents

when received. When set to false, the adaptor generates the

update notice to the dependents of the adaptor. The lazy dependency

mechanism avoids double-notificaton of dependents when the subject

does send updates.

subjectChannel <ValueModel> When this sends a change notice,

update the subject.

accessPath <SequencableCollection | nil> holds accessors to turn

the subject into the target

ProtocolAdaptor is an abstract class that provides its subclasses with

the ability to get and set an embedded value in an object other than

the application model, such as an instance variable in a domain model.

Each such adaptor has a subject, which is typically a composite domain

model, and specialized value-getting and -setting messages for extracting

the desired value from the subject.

For example, suppose you are creating a canvas containing one input

field for each part of a Customer object: accountNumber, name, company,

address, and so on. Since the accountNumber is held by a Customer

object rather than by the application model, an ordinary ValueHolder

offers no help in accessing it. While you can create a duplicate

accountNumber variable in the application model, and charge the

application model with the responsibility of updating the Customer

object whenever the input field is changed, this is cumbersome,

especially for a large number of such fields. A ProtocolAdaptor (in this

case, an AspectAdaptor) enables you to cut out the middle man by

getting and setting the accountNumber in the Customer object directly.

In this case, the Customer would be the subject of several

AspectAdaptors -- one adaptor translates #value & #value: into

#accountNumber & #accountNumber:, another adaptor manages the

customer name, and so on.

The subject can be changed during the life of an adaptor -- for

example, a new instance of Customer can become the focus of the adaptor's

inquiries. When a change of subject is likely, it is most economical

to first enfold the subject in a value holder. This value holder is known as a subject channel, because it provides a channel to the subject.

In that case, the adaptor would be created via a #subjectChannel: message rather than a #subject: message. In the example, instead of storing a Customer object in an instance variable of the application model, we would store a value holder containing the Customer object.

Because both the adaptor and its subject are capable of sending #update:with:from: messages to the same dependent, it is sometimes necessary to disable the adaptor's update facility. This is usually done at instance creation time, via a #subjectSendsUpdates: message.

By default, the adaptor assumes that the subject does not send redundant update messages.

ProtocolAdaptor is actually capable of extracting a value that is deeply embedded in the subject. For example, suppose the Customer holds an AccountNumber object, which holds an AccountPrefix object, which holds a prefixCharacter and a prefixNumber. The AspectAdaptor for the prefixNumber would need to send #accountNumber to the address, then send #accountPrefix to the account number. This series of messages is called the access path, and is initialized via #accessPath:.

AspectAdaptor is the most commonly used subclass of ProtocolAdaptor.

IndexedAdaptor is used to access an element in a collection.

When creating a subclass, equip it with the following methods:

setValueUsingTarget:to:

valueUsingTarget:

A subclass that implements #value: is responsible for sending an update message if its subject does not send one.

Available methods are:

accessPath: aSequenceableCollection

Answer a new instance of the receiver with accessPath aSequenceableCollection.

new

Create a new instance of the class.

subject: aSubject

Answer a new ProtocolAdaptor with a constant subject (aSubject). By default, the ProtocolAdaptor's subject does not send update notices to its dependents.

Note: For a ProtocolAdaptor which will change subjects frequently, or a group of ProtocolAdaptors which should all share a subject and change at the same time, subjectChannel: provides a convenient interface.

subject: aSubject accessPath: aSequenceableCollection

Create and initialize the ProtocolAdaptor.

subject: aSubject sendsUpdates: aBoolean

Answer a new ProtocolAdaptor with a constant subject (aSubject). This ProtocolAdaptor will send update messages when the value changes if aBoolean is false.

Note: For a ProtocolAdaptor which will change subjects frequently, or a group of ProtocolAdaptors which should all share a subject and change at the same time, subjectChannel:sendsUpdates: provides a convenient interface.

subject: aSubject sendsUpdates: aBoolean accessPath: aSequencableCollection

Create and initialize the ProtocolAdaptor.

subjectChannel: aValueModel

Answer a new ProtocolAdaptor with a variable subject (aValueModel is the subject channel) to notify it of changes in the subject. By default, the ProtocolAdaptor's subject does not send update notices to its dependents.

Note: A ProtocolAdaptor which will not change subjects does not need to use a subject channel. It is more efficient and convenient to set the subject using subject:

subjectChannel: aValueModel accessPath: aSequenceableCollection

Set our instance variables.

subjectChannel: aValueModel sendsUpdates: aBoolean

Answer a new ProtocolAdaptor with a variable subject (aValueModel is the subject channel) to notify it of changes in the subject. The ProtocolAdaptor will send update messages when the value changes if aBoolean is false.

Note: A ProtocolAdaptor which will not change subjects does not need to use a subject channel. It is more efficient and convenient to set the subject using subject:sendsUpdates:

subjectChannel: aValueModel sendsUpdates: aBoolean accessPath: aSequenceableCollection

Read the class description.

initialize

releaseParts

Remove the receiver as a dependent of the receiver's subject.

subject

Answer the current subject.

subject: anObject

Set the subject to be anObject. Send an update since the value has
probably changed too. If this ProtocolAdaptor has a subject channel,
delegate setting the new subject to it so that others depending
on the same subject channel value model will be informed automatically.
Note: For a ProtocolAdaptor which will change subjects frequently,
or a group of ProtocolAdaptors which should all share a subject and
change at the same time, subjectChannel: provides a convenient interface.

subjectChannel

Answer the ValueModel used to provide new subjects.

subjectChannel: aValueModel

Set or change the ValueModel we depend on to provide the latest subject.
In the rare cases where the subject channel needs to be reinitialized an
update message is sent on the assumption that the value has changed.

subjectSendsUpdates

Does our subject send updates to its dependents?

subjectSendsUpdates: aBoolean

Set or change the nature of the subject.
If the subject does not send updates, we
won't bother to depend on it.

accessPath

Answer the receiver's accessPath. This is a collection
of accessors used to turn the receiver's subject into
the target for messages.

accessPath: aSequenceableCollection

Set the receiver's accessPath to be aSequenceableCollection.
This will be used to turn the subject into the target.

setValue: newValue

Set a new value using the reciever's target.

target

Answer the receiver's target for operations.
If there is an accessPath it will hold accessors that
will be used to turn the subject into the target.

value

Answer the value returned by sending the receiver's
retrieval (get) selector to the receiver's target.

value: newValue

Set the currently stored value, and notify dependents.

valueUsingSubject: aSubject

Answer a value for the subject if aSubject were the

receiver's subject.

addDependent: anObject

Add anObject as one of the receiver's dependents.

removeDependent: anObject

Remove the argument, anObject, as one of the

receiver's dependents.

update: anAspect with: parameters from: anObject

If the update is from the subjectChannel, it must be because

there is a new subject.

isProtocolAdaptor

Answer as to whether the receiver transduces protocol

into ValueModel protocol.

printOn: aStream

printPathOn: aStream

access: anObject with: anAccessor

changedSubject

The subject has changed.

hookupToSubject

Add the receiver as a dependent of the receiver's subject.

makeAdaptorForRenderingStoreLeafInto: pair

renderingValueUsingSubject: aSubject

setSubject: anObject

Set the subject to be anObject. Send an update since the

value has probably changed too.

setValueUsingTarget: anObject to: newValue

Using anObject set a new value.

targetUsingSubject: aSubject

unhookFromSubject

Remove the receiver as a dependent of the receiver's subject.

valueUsingTarget: anObject

Answer the value returned by using anObject.

See Also, AspectAdaptor

IndexedAdaptor

1.11 IndexedAdaptor Class:

Added for V2.5+

Class IndexedAdaptor provides the appearance of a ValueHolder, but redirects the value and value: methods to the target by sending at: and at:put:, respectively.

When there is no target, the value is always nil; setValue: is a no-op; and value: only notifies the dependents.

Instance Variables:

index <Integer> The index to adapt.

An IndexedAdaptor is used to get and set an element in a collection. It is typically created by sending a #subject: message to this class, with the collection as the argument. It is then equipped with the index number or other lookup key of the desired element, via #forIndex:.

An IndexedAdaptor can manage a collection element that is embedded multiple levels within the subject, via an access path. It can also be told to withhold its update messages to avoid duplicating those sent by its subject. See ProtocolAdaptor for a fuller discussion of these abilities.

Avaialble methods are:

forIndex: anIndex

Create a new IndexedAdaptor and initialize the index to adapt.

The subject or subjectChannel and whether the subject sends updates must be initialized separately.

forIndex: anIndex accessPath: aSequencableCollection

Create a new IndexedAdaptor and initialize the index to adapt.

The subject or subjectChannel and whether the subject sends updates must be initialized separately.

forIndex

Answer the index we're adapting.

forIndex: anIndex

Set out index instance variable to anIndex.

setValueUsingTarget: anObject to: newValue

Set the value in anObject using at:put:

valueUsingTarget: anObject

Answer the value returned by sending anObject with at:

update: anAspect with: aParameter from: anObject

Update our dependents or ask our parent to pass on the message.

printOn: aStream

Print ourself to aStream.

See Also, SlotAdaptor

1.12 SlotAdaptor Class:

Added for V2.5+

Class SlotAdaptor redirects the value and value: methods to the

target by sending instVarAt: and instVarAt:put:, respectively.

When there is no target, the value is always nil; setValue: is a

no-op; and value: only notifies the dependents.

The subject is assumed to not send updates. However, if it uses

#at as the aspect and provides the index as the parameter, it will

be treated as a change notice an propogated to the dependents of

the SlotAdaptor.

Available methods are:

forIndex: instVarIndex

Set out index to instVarIndex.

setValueUsingTarget: anObject to: newValue

Set the value in anObject using instVarAt:put:

valueUsingTarget: anObject

Answer the value returned by sending anObject instVarAt:

printOn: aStream

Print our index to aStream.

See Also, IndexedAdaptor

1.13 ValueHolder Class:

Added for V2.5+

ValueHolder is a very simple Model , no more than a value holder with updates.

Instance variables:

value <Object> the current value

A ValueHolder is the simplest value model. It merely holds a value,

and notifies the dependents of that value whenever it is changed.

A ValueHolder is widely used to enfold the strings, numbers and

other data objects that are displayed in widgets. For this reason,

every object has been made capable of enfolding itself in a ValueHolder

when it receives an #asValue message. ValueHolder also provides both

general and specialized creation messages for enfolding a given value.

Available methods are:

newBoolean

Answer a new instance, initialized to false.

newFraction

Answer a new instance, initialized to 0.0.

newString

Answer a new instance, initialized to an empty string.

with: initialValue

Answer a new instance, initialized to the given value.

setValue: aValue

Just initialize the value without notifying dependents of a change.

value

Return the current stored value.

printOn: aStream

Let what is printed reflect the value of the receiver.

1.14 BufferedValueHolder Class

Added for V2.5+

Class BufferedValueHolder is a wrapper for a ValueModel (the

subject). Clients see the current value of the subject until

value: provides a new value. The new value is not provided to

the subject until the application directs it via a setting the

triggerChannel value to true. The buffered value may be discarded

by setting the trigger channel value to false.

Instance Variables:

subject <ValueModel> The ultimate source/destination of the value.

triggerChannel <ValueModel> When this changes, push the current

value down to the subject. If the

value is equal to notYetAssigned, do nothing.

Class Variables (see BVHGlobalVar class):

notYetAssigend <Object> A distinguished value used to

indicate that value has not been set.

A BufferedValueHolder is used to hold a temporary copy of the

value in another valueModel (known as the subject). The

application modifies the temporary copy, but the BufferedValueHolder only gives this temporary value to its subject when

the application confirms the changes. The application also has

the option of canceling the changes, resetting the temporary

copy to the subject's value.

For example, suppose the application provides a series of input

fields for entering customer name, address, phone, etc., but we

only want the Customer object to be updated after the user has finished entering data and has indicated completion by clicking on an OK button. This technique is often used in database applications, to postpone updating the customer record in the database until all changes to that record are completed. In this application, the customer's old address would likely be held by an AspectAdaptor on the Customer object. The aspect adaptor would become the subject of a BufferedValueHolder. The BufferedValueHolder would make a temporary copy of the customer's address and make that value available to the input field for editing.

The user could change the address, but so far only the temporary copy has been altered. Only when the users clicks on 'OK' does the application notify each field's BufferedValueHolder to replace the corresponding value in the Customer object.

A BufferedValueHolder is created by sending a #subject:triggerChannel: message to this class. The subject is a valueModel containing the data value. The triggerChannel is a ValueHolder containing the boolean object false. Later, when the user clicks on 'OK', the application can cause the temporary copy to become the subject's value by setting the triggerChannel's value to true. The application can also cancel any edits, by setting the triggerChannel's value to false. Note that the prior value in the triggerChannel is not significant -- setting the value to true when it is already true has the same effect as if it were previously false.

By using the same triggerChannel for all of the BufferedValueHolders, the application can cause them all to be updated at the same time. This is the usual arrangement for a set of related widgets.

This class exists because Little Smalltalk does not have Class variables available. The following class is used to store the Class variable notYetAssigned:

Class BVHGlobalVar methods you need to use are:

notYetAssigned

Return the stored variable.

notYetAssigned: aBoolean

Set/Reset the stored variable.

The Methods for the BufferedValueHolder Class are:

subject: aSubject triggerChannel: aTrigger

Create a new BufferedValueHolder which provides buffering for the ValueModel aSubject, and which pushes the buffered

value into the subject when the ValueModel aTrigger changes.

initialize

Setup this BufferValueHolder instance.

releaseParts

Remove the receiver as dependents of the triggerChannel

and subject.

subject

The subject of our adapting logic. The saved value will

be sent to the subject when the trigger channel indicates

that it is time to do so.

subject: aValueModel

The subject is the actual respository for the value

held in this object.

triggerChannel

The object we depend on which sends an update message

when the saved value should be inserted into the subject.

triggerChannel: aValueModel

An object to depend on which will send an update message

to trigger the copy of the saved value to the subject.

value

Answer with the current value.

valueUsingSubject: aSubject

ˆ subject valueUsingSubject: aSubject

addDependent: anObject

Add anObject as one of the receiver's dependents.

removeDependent: anObject

Remove the argument, anObject, as one of the receiver's dependents.

changedTrigger

Process the trigger notification. "

Unhooking and rehooking the subject prevents

dependency notification being propogated thru this object.

The dependents of this object have already been informed

of the current value.

There's nothing to do if no new value has been set

since the last trigger.

hookupToSubject

Add the receiver as a dependent of the receiver's subject.

renderingValueUsingSubject: aSubject

Return the value using aSubject.

unhookFromSubject

Remove the receiver as a dependent of the receiver's subject.

update: anAspect with: parameters from: anObject

isBuffering

Answer true if a value is being buffered

See Also, ValueHolder

1.15 AspectAdaptor Class:

Added for V2.5+

Class AspectAdaptor provides the appearance of a ValueHolder, but

redirects the value and value: methods to the target by sending

getSelector and putSelector, respectively. The putSelector is assumed

to take a single argument which is the argument provided to value:.

When there is no target, the value is always nil; setValue: is a no-op;

and value: only notifies the dependents.

Instance Variables:

getSelector <Symbol> 0-arg message selector

putSelector <Symbol> 1-arg message selector

aspect <Symbol | nil> aspect for which the adaptor is willing to

field updates. If nil, use the getSelector instead

An AspectAdaptor is widely used in applications, to get and set an

embedded value. While a ValueHolder typically manages a value held by

an application model, an AspectedAdaptor typically manages a value held

by a domain model, which itself is held by the application model. The

domain model is the adaptor's subject, and the adaptor must be equipped

with messages (getSelector and putSelector) for accessing the desired value

in the subject. See ProtocolAdaptor for a descriptive

example.

An AspectAdaptor is typically created by sending a #subject: message to

this class, with the domain model as the argument. The getSelector and

putSelector are typically the same message (with a colon, in the case of

the putSelector) and can be set via #forAspect:. When the getSelector and

putSelector are dissimilar, use #accessWith:assignWith: to set them.

An AspectAdaptor can manage a value that is embedded multiple levels within

the subject, via an access path. It can also be told to withhold its

update messages to avoid duplicating those sent by its subject. See

ProtocolAdaptor for a fuller discussion of these abilities.

Available methods are:

accessWith: getSymbol assignWith: putSymbol

Create a new AspectAdaptor and initialize the getSelector and putSelector

with getSymbol and putSymbol, respectively. The subject or subjectChannel

and whether the subject sends updates must be initialized separately.

accessWith: getSymbol assignWith: putSymbol accessPath: aSequencableCollection

Create a new AspectAdaptor and initialize the getSelector and putSelector

with getSymbol and putSymbol, respectively. The subject or subjectChannel

and whether the subject sends updates must be initialized separately.

forAspect: anAspectSymbol

Create a new AspectAdaptor and initialize the getSelector and putSelector

based on anAspectSymbol and the symbol with a colon appended.

The subject or subjectChannel and whether the subject sends updates must

be initialized separately.

forAspect: anAspectSymbol accessPath: aSequencableCollection

Create a new AspectAdaptor and initialize the getSelector and putSelector

based on anAspectSymbol and the symbol with a colon appended.

The subject or subjectChannel and whether the subject sends updates must

be initialized separately.

accessWith: getSymbol assignWith: putSymbol

Set or change the symbols used to access the subject.

accessWith: getSymbol assignWith: putSymbol aspect: aspectSymbol

Set or change the symbols used to access the subject.

forAspect

Answer the aspect we're adapting.

forAspect: anAspectSymbol

Set or change the symbols used to access the subject.

initialize

Initialize this instance of the class.

setValueUsingTarget: anObject to: newValue

Set the value of anObject by sending the receiver's

store (put) selector to the anObject with argument newValue.

valueUsingTarget: anObject

Answer the value returned by sending the receiver's

retrieval (get) selector to anObject.

update: anAspect with: parameter from: sender

Propagate change if the sender is the receiver's subject

and anAspect is the receiver's aspect.

printOn: aStream

Output the object onto aStream.

See Also, ProtocolAdaptor

1.16 Pen Class:

The class Pen is a class that opens a Window for performing simple

graphics commands in. This class has been re-written & is completely

different from the intentions of the Little SmallTalk author, Tim Budd.

Instead of using a plotting device (How many of those are there for the

Amiga?), this class simply opens a Window that can be used to see the

results of the Pen methods.

NOTE: There's a limit of 20 for how many Plot Windows can be open

at the same time. AmigaTalk will tell you via Requesters when

this limit is violated.

This class responds to the following methods:

new

make a new instance of class Pen, initializing the

instance variables (default title: 'Unknown Plot').

new: newPlotTitle

make a new instance of class Pen, initializing the

instance variables & using the supplied newPlotTitle as

the Plot Window title.

openPlotEnv: sizePoint

Open the Plot Window with the given size (sizePoint is of class Point ,

so (sizePoint x) is the width, & (sizePoint y) is the height of the

Plot Window).

WARNING: You can only open a Plot Window as big as the AmigaTalk screen

(default 640 by 480).

closePlotEnv: whichPlotTitle

Close the Plot Window with the given title.

movePlotEnvBy: deltaPoint

Move the Plot Window by the given deltaPoint amounts (deltaPoint is

of class Point , so (deltaPoint x) is x movement,

& (deltaPoint y) is y movement of the Plot Window.

WARNING: There is no bounds checking for this, so make sure you keep

the Plot Window visible!

setLineType: bitPattern

Change the type of the line to plot with to the given bitPattern value.

(example: 2r11110000111100001111000011110000 = 16rF0F0F0F0 will draw

a dashed line). This is equivalent to SetDrPt() in graphics.library.

drawText: text at: startPoint

Place the given text at the given starting point using the current pen colors.

WARNING: There is no bounds checking for this, so make sure you keep

the text inside the Plot Window!

drawBox: fromPoint to: endPoint

Draw a box (fromPoint x) @ (fromPoint y)

to (endPoint x) @ (endPoint y). This is different from the

graphics.library DrawBox() call in that the endPoint is NOT interpreted

to be the width & height of the box. If you want to use the second

point as width @ height, simply add this:

endPoint x <- fromPoint x + endPoint x.

endPoint y <- fromPoint y + endPoint y.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

drawCircleAt: centerPoint radius: r

Draw a circle at the given centerPoint with the given radius using

the current pen colors.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

circleRadius: radius

Draw a circle at the current location, with the given radius using

the current pen colors.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

drawArcAround: pivotPoint for: angleSizer

Draw an arc starting at the current location, around the given pivotPoint for the

given angleSize (expressed in Radians) using the current pen colors.

This method is math intensive, so don't expect it to be fast!

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

drawArcAt: startPoint around: pivotPoint for: angleSize

Draw an arc starting at startPoint, around the given pivotPoint for the

given angleSize (expressed in Radians) using the current pen colors.

This method is math intensive, so don't expect it to be fast!

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

drawTo: endPoint

Draw a line from the current location to the given endPoint using the

current pen colors.

WARNING: There is no bounds checking for this, so make sure you keep inside the Plot Window!

goTo: aPoint

Move the drawing point to the given aPoint.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

drawLine: fromPoint to: endPoint

Draw a line fromPoint to endPoint using the current pen colors.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

drawPoint: atPoint

Draw a pixel atPoint using the current pen colors.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

direction

This method returns a Radian value, indicating the current direction

that the Pen will go with the go: method.

direction: radianAngle

Set the direction that the Pen will go with the go: method.

erase

Fill the Plot Window with the background color & erase all Plotting.

extent

Return a Point that indicates the width @ height of the Plot Window.

location

Return a Point that indicates the x @ y of the

plotter's location.

center

Move the current plotting location to the center of the Plot Window.

tellPens

Return a Point that indicates the fpen @ bpen

of the Plot Window.

setPens: penSet

Change the fpen @ bpen values to (penSet x) @ (penSet y) respectively.

go: anAmount

Move the plotting location anAmount in the current direction.

anAmount is a scalar value (Integer or Float).

turn: addedAngle

Change the current direction by the given addedAngle (in Radians).

titleIs

Return a String that corresponds to the title of the plot window.

SEE ALSO FormPen , SavePen , ShowPen

1.17 FormPen Class:

The class FormPen is a sub-class of Pen that allows the User

to put together a collection (actually a Bag) of lines.

This class responds to the following methods:

new

Initialize the FormPen class instance.

add: startingPoint to: endPoint

Add a line with the given points to the instance.

with: aPen displayAt: location

Draw all the lines contained in the FormPen using the given aPen.

aPen is of class Pen .

1.18 SavePen Class:

The class SavePen is a sub-class of FormPen that allows the User

to save a drawing made by a Pen. What the original author of

this class means by save isn't quite clear.

This class responds to the following methods:

setForm: aForm

Initialize the instance variable with aForm of class Form .

goTo: aPoint

Add a line from the current location to aPoint of class Point

to aForm.

1.19 ShowPen Class:

The class ShowPen is a sub-class of Pen that allows the User

to see some fancy uses of the Pen class.

This class responds to the following methods:

withPen: aPen

Initialize the instance variable(s) (aPen is of class Pen .

poly: nSides length: length

Draw a ploygon with the given number of sides each with the given

length.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window! Also, there is no such thing as a

ploygon with less than 3 sides, but this method doesn't

perform any check for this!

spiral: n angle: a

Draw a spiral with the given number of segments (which is also the
length of the segments), changing the direction angle by a Radians .
WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!

1.20 Form Class:

The class Form is a sub-class of Object that allows the
User to draw figures using ASCII text. This class is NOT ported to
the graphic capabilities of the Amiga, so don't expect to get any useful
pictures with it. I've just left the Smalltalk code as descriptions
of what the methods actually do. Use class Pen or the Curses
primitives (in AmigaTalk:User/Curses.st) for drawing simple
pictures instead.

This class responds to the following methods:

new

Initialize the instance of Form.

clipFrom: upperLeft to: lowerRight

"You figure it out:"

```
! newForm newRow rsize left top rText !
left <- upperLeft y - 1. " left hand side"
top <- upperLeft x - 1.
rsize <- lowerRight y - left.
newForm <- Form new.
(upperLeft x to: lowerRight x)
do: [:i |
newRow <- String new: rsize.
rText <- self row: i.
(1 to: rsize)
do: [:j |
newRow at: j
put: (rText at: (left + j)
ifAbsent: [$ ])
].
newForm row: (i - top) put: newRow
].
ˆ newForm
```

columns

```
ˆ text inject: 0 into: [:x :y | x max: y size ]
display
smalltalk clearScreen.
self printAt: 1 @ 1.
’ ’ printAt: 20 @ 0
eraseAt: aPoint ! location !
location <- aPoint copy.
text do: [:x | (String new: (x size)) printAt: location.
location x: (location x + 1) ]
extent
ˆ self rows @ self columns
first
ˆ text first
next
ˆ text next
overLayForm: sourceForm at: startingPoint
! newRowNum rowText left rowSize !
newRowNum <- startingPoint x.
left <- startingPoint y - 1.
sourceForm do: [:sourceRow |
rowText <- self row: newRowNum.
rowSize <- sourceRow size.
rowText <- rowText padTo: (left + rowSize).
(1 to: rowSize) do: [:i |
((sourceRow at: i) ~= $ )
ifTrue: [ rowText at: (left + i)
put: (sourceRow at: i)]].
self row: newRowNum put: rowText.
newRowNum <- newRowNum + 1]
placeForm: sourceForm at: startingPoint
! newRowNum rowText left rowSize !
newRowNum <- startingPoint x.
left <- startingPoint y - 1.
sourceForm do: [:sourceRow |
rowText <- self row: newRowNum.
rowSize <- sourceRow size.
rowText <- rowText padTo: (left + rowSize).
(1 to: rowSize) do: [:i |
rowText at: (left + i)
put: (sourceRow at: i)].
self row: newRowNum put: rowText.
newRowNum <- newRowNum + 1]
reversed ! newForm columns newRow !
columns <- self columns.
newForm <- Form new.
(1 to: self rows) do: [:i |
newRow <- text at: i.
newRow <- newRow ,
(String new: (columns - newRow size)).
newForm row: i put: newRow reversed ].
ˆ newForm
rotated ! newForm rows newRow !
rows <- self rows.
newForm <- Form new.
(1 to: self columns) do: [:i |
newRow <- String new: rows.
(1 to: rows) do: [:j |
newRow at: ((rows - j) + 1)
put: ((text at: j)
at: i ifAbsent: [$ ])].
newForm row: i put: newRow ].
ˆ newForm
row: index
ˆ text at: index ifAbsent: [”]
row: index put: aString
(index > text size)
ifTrue: [ [text size < index] whileTrue:
[text <- text grow: ”] ].
text at: index put: aString
rows
ˆ text size
printAt: aPoint ! location !
location <- aPoint copy.
text do: [:x | x printAt: location.
location x: ((location x) + 1) ]
```

1.21 UndefinedObject Class:

The pseudo variable nil is an instance (usually the only instance)

of the class UndefinedObject. nil is used to represent undefined

values, and is also typically returned in error situations. nil is also

used as a terminator in sequences, as for example in response to the

message next when there are no further elements in a sequence.

Examples: Printed result:

nil isNil True

This class responds to the following methods:

isNil

Overrides method found in Object. Return true.

notNil

Overrides method found in Object. Return false.

printString

Return ’nil’.

1.22 Symbol Class:

Instances of the class Symbol are created either by their literal

representation, which is a pound sign followed by a string of nonspace

characters (for example #aSymbol), or by the message asSymbol being

passed to an object. Symbols cannot be created using new. Symbols

are guaranteed to have unique representations; that is, two symbols

representing the same characters will always test equal to each other.

Inside of literal arrays, the leading pound signs on symbols can be

eliminated, for example: #(these are symbols).

Examples: Printed result:

#abc == #abc True

#abc == #ABC False

#abc ~~ #ABC True

#abc printString #abc

’abc’ asSymbol #abc

#do:ifAbsent: numArgs 2

This class responds to the following methods:

==

Return true if the two symbols represent the same characters, false

otherwise.

asString

Return a String representation of the symbol without the
leading pound sign.
printString
Return a String representation of the symbol, including the
leading pound sign.
numArgs
Return the number of arguments that the receiver would require
if it were to be interpreted as a message.

1.23 Boolean Class:

The class Boolean provides protocol for manipulating true and false values. The pseudo-variables true and false are instances of the subclasses of Boolean; True and False, respectively. The subclasses True and False, in combination with blocks, are used to implement conditional control structures. Note, however, that the bytecodes may optimize conditional tests by generating code in-line, rather than using message passing. Note that bit-wise boolean operations are provided by class Integer.
Examples: Printed result:
(1 > 3) & (2 < 4) False
(1 > 3) | (2 < 4) True
(1 > 3) and: [2 < 4] False
This Class responds to the following methods:
&
The argument must be a boolean. Return the logical conjunction (and)
of the two values.
|
The argument must be a boolean. Return the logical disjunction (or)
of the two values.
and: aBlock
The argument must be a block. Return the logical conjunction (and)
of the two values. If the receiver is false the second argument
is not used, otherwise the result is the value yielded in evaluating
the argument block.
or: aBlock
The argument must be a block. Return the logical disjunction (or)
of the two values. If the receiver is true the second argument
is not used, otherwise the result is the value yielded in evaluating

the argument block.

eqv: aBoolean

The argument must be a boolean. Return the logical equivalence (eqv) of the two values.

xor: aBoolean

The argument must be a boolean. Return the logical exclusive or (xor) of the two values.

1.24 True Class:

The pseudo-variable true is an instance (usually the only instance) of the class True.

Examples: Printed result:

(3 < 5) not False

(3 < 5) ifTrue: [17] 17

This Class responds to the following methods:

ifTrue: trueAlternativeBlock

Return the result of evaluating the argument block.

ifFalse: falseAlternativeBlock

Return nil.

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

Return the result of evaluating the first argument block.

ifFalse: falseAlternativeBlock ifTrue: trueAlternativeBlock

Return the result of evaluating the second argument block.

not

Return false.

1.25 False Class:

The pseudo-variable false is an instance (usually the only instance) of the class False.

Examples: Printed result:

(1 < 3) ifTrue: [17] 17

(1 < 3) ifFalse: [17] nil

This Class responds to the following methods:

ifTrue: trueAlternativeBlock

Return nil.

ifFalse: falseAlternativeBlock

Return the result of evaluating the argument block.

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

Return the result of evaluating the second argument block.

ifFalse: falseAlternativeBlock ifTrue: trueAlternativeBlock

Return the result of evaluating the first argument block.

not

Return true.

1.26 Magnitude Class:

The class Magnitude provides protocol for those subclasses possessing a linear ordering. For the sake of efficiency, most subclasses redefine some or all of the relational messages. All methods are defined in terms of the basic messages <, = and >, which are in turn defined circularly in terms of each other. Thus each subclass of Magnitude must redefine at least one of these messages.

Examples: Printed result:

$A max: $a $a

4 between: 3.1 and: (17/3) True

This Class responds to the following methods:

<

Relational less than test. Returns a boolean.

<=

Relational less than or equal test.

=

Relational equal test. Note that this differs from ==, which is an object equality test.

~=

Relational not equal test, opposite of =.

>=

Relational greater than or equal test.

>

Relational greater than test.

between: low and: high

Relational test for inclusion.

max: arg

Return the maximum of the receiver and argument value.

min: arg

Return the minimum of the receiver and argument value.

1.27 Char Class:

This class defines protocol for objects with character values.

Characters possess an ordering given by the underlying representation,

however arithmetic is not defined for character values. Characters are

written literally by preceding the character desired with a dollar sign,

for example: $a $B $$.

Examples: Printed result:

$A < $0 False

$A asciiValue 65

$A asString A

$A printString $A

$A isVowel True

$A digitValue 10

This Class responds to the following methods:

==

Object equality test. Two instances of the same character always

test equal.

asciiValue

Return an Integer representing the ASCII value of the receiver.

asLowercase

If the receiver is an uppercase letter returns the same letter in

lowercase, otherwise returns the receiver.

asUppercase

If the receiver is a lowercase letter returns the same letter in

uppercase, otherwise returns the receiver.

asString

Return a length one string containing the receiver. Does not contain

leading dollar sign, compare to printString.

digitValue

If the receiver represents a number (for example $9) return

the digit value of the number. If the receiver is an uppercase

letter (for example $B) return the position of the number in

the uppercase letters + 10, ($B returns 11, for example). If

the receiver is neither a digit nor an uppercase letter an error is

given and nil returned.

isAlphaNumeric

Respond true if receiver is either digit or letter, false otherwise.

isDigit

Respond true if receiver is a digit, false otherwise.

isLetter

Respond true if receiver is a letter, false otherwise.

isLowercase

Respond true if receiver is a lowercase letter, false otherwise.

isSeparator

Respond true if receiver is a space, tab or newline, false otherwise.

isUppercase

Respond true if receiver is an uppercase letter, false otherwise.

isVowel

Respond true if receiver is $a, $e, $i, $o or $u, in either upper or lower case.

printString

Respond with a string representation of the character value. Includes leading dollar sign, compare to asString, which does not include $.

1.28 Number Class:

The class Number is an abstract superclass for Integer and Float. Instances of Number cannot be created directly. Relational messages and many arithmetic messages are redefined in each subclass for arguments of the appropriate type. In general, an error message is given and nil returned for illegal arguments.

Examples: Printed result:

3 < 4.1 True

3 + 4.1 7.1

3.14159 exp 23.1406

9 gamma 40320

5 reciprocal 0.2

0.5 radians 0.5 radians

13 roundTo: 5 15

13 truncateTo: 5 10

This Class responds to the following methods:

maxtype: aNumber

Return the receiver if the receiver has greater generality than the argument, otherwise return the argument coerced into being the same type as the receiver.

= aNumber

Compare the Receiver with the argument, return true if they are the same type, false otherwise.

< aNumber

Return true if the Receiver has less generality than the argument, false otherwise.

> aNumber

Return true if the Receiver has greater generality than the argument, false otherwise.

+ aNumber

Mixed type addition.

- aNumber

Mixed type subtraction.

* aNumber

Mixed type multiplication

/ aNumber

Mixed type division.

ˆ aNumber

Exponentiation, same as raisedTo:.

@ aNumber

Construct a point with coordinates being the receiver and the argument.

abs

Absolute value of the receiver.

exp

e raised to the power represented by the receiver.

gamma

Return the gamma function (generalized factorial) evaluated at the receiver.

ln

Natural logarithm of the receiver.

log: aNumber

Logarithm in the given base.

negated

The arithmetic inverse of the receiver.

negative

True if the receiver is negative.

pi

Return the approximate value of the receiver multiplied by (3.1415926).

positive

True if the receiver is positive (>= 0).

radians

Argument converted into radians.

raisedTo: aNumber

The receiver raised to the argument value.

reciprocal

The arithmetic reciprocal of the receiver.

roundTo: aNumber

The receiver rounded to units of the argument (see the source in

AmigaTalk:General/Number.st).

sign

Return -1, 0 or 1 depending upon whether the receiver is negative,

zero or positive, respectively.

sqrt

Square root. nil if receiver is less than zero.

squared

Return the receiver multiplied by itself.

strictlyPositive

True if the receiver is greater than zero.

to: highValue

Interval from Receiver to argument value (highValue) with step of 1.

to: highValue by: stepSize

Interval from Receiver to argument (highValue) in given steps.

truncatedTo: aNumber

The receiver truncated to units of the argument. (see the source in

AmigaTalk:General/Number.st).

1.29 Integer Class:

The class Integer provides protocol for objects with integer values.

Examples: Printed result:

5 + 4 7

5 allMask: 4 True

4 allMask: 5 False

5 anyMask: 4 True

5 bitAnd: 3 1

5 bitOr: 3 7

5 bitInvert -6

254 radix: 16 16rFE

-5 // 4 -2

-5 quo: 4 -1

-5 intNegRem: 4 1

-5 rem: 4 -1

8 factorial 40320

This Class responds to the following methods:

= aNumber

Return true if the Receiver & the argument are equal, false otherwise.

> aNumber

Return true if the Receiver is greater than the argument, false otherwise.

< aNumber

Return true if the Receiver is less than the argument, false otherwise.

+ aNumber

Return the sum of the Receiver & the argument.

- aNumber

Return the difference between the Receiver & the argument.

* aNumber

Return the product of the Receiver & the argument.

/ aNumber

Return the quotient of the Receiver & the argument.

// aNumber

Integer quotient, truncated towards negative infinity (compare to quo:).

intNegRem: aNumber

Integer remainder, truncated towards negative infinity (compare to rem:).

allMask: anInteger

Argument must be Integer. Treating receiver and argument as bit strings, return true if all bits with 1 value in argument correspond to bits with 1 values in the receiver.

anyMask: anInteger

Argument must be Integer. Treating receiver and argument as bit strings, return true if any bit with 1 value in argument corresponds to a bit with value 1 in the receiver.

asCharacter

Return the Char with the same underlying ASCII representation as the low order eight bits of the receiver.

asFloat

Return a floating point value with same magnitude as receiver.

asHex

Return the Receiver as a HexaDecimal String .

asBinary

Return the Receiver as a binary String .

asOctal

Return the Receiver as an octal String .

bitAnd: anInteger

Argument must be Integer. Treating the receiver and argument as bit strings, return logical and of values.

bitAt: anInteger

Argument must be Integer greater than 0 and less than underlying word size. Treating receiver as a bit string, return the bit value at the given position, numbering from low order (or rightmost) position.

bitInvert

Return the receiver with all bit positions logically inverted.

bitOr: anInteger

Return logical or of values.

bitShift: anInteger

Treating the receiver as a bit string, shift bit values by amount indicated by anInteger. Negative values shift right, positive left.

bitXor: anInteger

Return logical xor of values.

even

Return true if receiver is even, false otherwise.

factorial

Return the factorial of the receiver. Return is a Float for large numbers.

gcd: anInteger

Argument must be Integer. Return the greatest common divisor of the receiver and argument.

highBit

Return the location of the highest 1 bit in the receiver. Return nil if the Receiver is zero.

lcm: anInteger

Argument must be Integer. Return least common multiple of receiver and argument.

noMask: anInteger

Argument must be Integer. Treating receiver and argument as bit strings, return true if no 1 bit in the argument corresponds to a 1

bit in the receiver.

odd

Return true if receiver is odd, false otherwise.

quo: anInteger

Return quotient of Receiver divided by argument.

radix: aNumber

Return a string representation of the receiver value, printed in the

base represented by aNumber. aNumber value must be <= 36

and >= to 2.

asSignedHex

Same as asHex only the Integer is treated as a signed quantity.

asSignedBinary

Same as asBinary only the Integer is treated as a signed quantity.

asSignedOctal

Same as asOctal only the Integer is treated as a signed quantity.

rem: anInteger

Remainder after receiver is divided by argument value.

timesRepeat: aBlock

Repeat argument block the number of times given by the receiver.

1.30 LongInteger Class:

LongInteger Class is for 64-Bit integer representation.

Since there are four functions in utility.library that

produce 64-bit quantities, I felt that a separate Class

should make use of them.

signed32BitDivide is really the SDivMod32() function.

unsigned32BitDivide is really the UDivMod32() function.

signed64BitMultiply is really the SMult64() function.

unsigned64BitMultiply is really the UMult64() function.

NOTE: Primitives for addition & subtraction will be added later.

Methods are:

= aNumber

Return true if the receiver is equal to aNumber.

> aNumber

Return true if the receiver is greater than aNumber.

< aNumber

Return true if the receiver is less than aNumber.

asString

Return the receiver as a String Object.

asFloat

Return the receiver as a Float Object.

even

Return true if the receiver is an even number.

odd

Return true if the receiver is an odd number.

getLower32Bits

Return the lower 32 Bits of the receiver.

getUpper32Bits

Return the upper 32 Bits of the receiver.

signed32BitDivide: dividend by: divisor

Perform some LongInteger signed division.

dividend & divisor are 32-Bit Integers, upper32Bits is

really the Quotient & lower32Bits is really the Remainder.

unsigned32BitDivide: dividend by: divisor

Perform some LongInteger unsigned division.

dividend & divisor are 32-Bit Integers, upper32Bits is

really the Quotient & lower32Bits is really the Remainder.

signed64BitMultiply: arg1 times: arg2

Evaluate a signed 64-bit product. arg1 & arg2 are

NOT necessarily 64-bit Integers.

unsigned64BitMultiply: arg1 times: arg2

Evaluate an unsigned 64-bit product. arg1 & arg2 are

NOT necessarily 64-bit Integers.

quotientIs

Return the Quotient of a signed/unsigned32BitDivide: method.

remainderIs

Return the Remainder of a signed/unsigned32BitDivide: method.

1.31 Float Class:

The class Float provides protocol for objects with floating point values.

Examples: Printed result:

4.2 * 3 12.6

2.1 |^ 4 19.4481

2.1 raisedTo: 4 19.4481

0.5 arcSin 0.523599 radians

2.1 reciprocal 0.47619

4.3 sqrt 2.07364

This Class responds to the following methods:

= aNumber

Return true if the Receiver & the argument have the same value, false otherwise.

< aNumber

Return true if the receiver is less than the argument.

> aNumber

Return true if the receiver is greater than the argument.

+ aNumber

Return the sum of the Receiver & the argument.

- aNumber

Return the difference of the Receiver & the argument.

* aNumber

Return the product of the Receiver & the argument.

/ aNumber

Return the quotient of the Receiver & the argument.

ˆ aNumber

Floating point exponentiation.

arcCos

Return a Radian representing the arcCos of the receiver.

arcSin

Return a Radian representing the arcSin of the receiver.

arcTan

Return a Radian representing the arcTan of the receiver.

asFloat

Return the receiver.

ceiling

Return the Integer ceiling of the receiver.

coerce: aNumber

Convert the argument into being type Float.

exp

Return e raised to the receiver value.

floor

Return the Integer floor of the receiver.

fractionPart

Return the fractional part of the receiver.

gamma

Return the value of the gamma function applied to the receiver value.

integerPart

Return the integer part of the receiver.

ln

Return the natural log of the receiver.

radix: aNumber

Return a string containing the printable representation of the receiver in the given radix. Argument must be an Integer <= 36 and >= 2.

rounded

Return the receiver rounded to the nearest integer.

sqrt

Return the square root of the receiver.

truncated

Return the receiver truncated to the nearest integer.

1.32 Radian Class:

The class Radian is used to represent radians. Radians are a unit of measurement, independent of other numbers. Only radians will respond to the trigonometric functions such as sin & cos. Numbers can be converted into radians by passing them the message radians. Similarly, radians can be converted into numbers by sending them the message asFloat. Notice that only a limited range of arithmetic operations are permitted on Radians. Radians are normalized to be between 0 and 2 * pi .

Examples: Printed result:

0.5236 radians sin 0.5

0.5236 radians cos 0.866025

0.5236 radians tan 0.577352

0.5 arcSin asFloat 0.523599

This Class responds to the following methods:

new: x

Create a new instance of Class Radian from x normalized to between 0 & 2 * pi.

< arg

Return true if the Receiver is less than the argument.

= arg

Return true if the argument is equal to the Receiver.

asFloat

Return the receiver as a floating point number.

cos

Return a floating point number representing the cosine of the receiver.

sin

Return a floating point number representing the sine of the receiver.

tan

Return a floating point number representing the tangent of the receiver.

printString

Display the Reciever as a String in the Status Window.

1.33 Point Class:

Points are used to represent pairs of quantities, such
as coordinate pairs.

Examples: Printed result:

(10@12) < (11@14) True

(10@12) < (11@11) False

(10@12) max: (11@11) 11@12

(10@12) min: (11@11) 10@11

(10@12) dist: (11@14) 2.23607

(10@12) transpose 12@10

This Class responds to the following methods:

< aPoint

True if both values of the receiver are less than the corresponding
values in the argument.

<= aPoint

True if the first value is less than or equal to the corresponding
value in the argument, and the second value is less than the
corresponding value in the argument.

>= aPoint

True if both values of the receiver are greater than or equal to the
corresponding values in the argument.

* scale

Return a new point with coordinates multiplied by the argument value.

/ scale

Return a new point with coordinates divided by the argument value.

// scale

Return a new point with coordinates divided by the argument value.

+ delta

Return a new point with coordinates offset by the corresponding values in the argument.

abs

Return a new point with coordinates having the absolute value of the receiver.

dist: aPoint

Return the Euclidean distance between the receiver and the argument point.

max: aPoint

The argument must be a Point. Return the lower right corner of the rectangle defined by the receiver and the argument.

min: aPoint

The argument must be a Point. Return the upper left corner of the rectangle defined by the receiver and the argument.

transpose

Return a new point with coordinates being the transpose of the receiver.

x

Return the first coordinate of the receiver.

x: aValue

Set the first coordinate of the receiver.

x: xValue y: yValue

Sets both coordinates of the receiver.

y

Return the second coordinate of the receiver.

y: aValue

Set the second coordinate of the receiver.

1.34 Random Class:

The class Random provides protocol for random number generation. Sending the message next to an instance of Random results in a Float between 0.0 and 1.0, randomly distributed. By default, the pseudo-random sequence is the same for each object in class Random. This can be altered using the message "randomize".

Examples: Printed result:

i <- Random new

i next 0.759

i next 0.157

i next: 3 #(0.408 0.278 0.547)

i randInteger: 12 5

i between: 4 and: 17.5 10.0

This Class responds to the following methods:

new

Initialize the seed Object to 1.

between: low and: high

Return a random number uniformly distributed between the two arguments.

first

Return a random number between 0.0 and 1.0. This message merely
provides consistency with protocol for other sequences, such as
Arrays or Intervals.

next

Return a random number between 0.0 and 1.0.

next: n

Return an Array containing the next n random numbers, where
n is the argument value.

randInteger: limit

The argument must be an Integer. Return a random integer
between 1 and the value given.

randomize

Change the pseudo-random number generator seed by a time dependent
value.

1.35 Collection Class:

The class Collection provides protocol for groups of objects, such as
Arrays or Sets. The different forms of collections are distinguished
by several characteristics, among them whether the size of the collection
is fixed or unbounded, the presence or absence of an ordering, and their
insertion or access method. For example, an Array is a collection with
a fixed size and ordering, indexed by integer keys. A Dictionary, on
the other hand, has no fixed size or ordering, and can be indexed by
arbitrary elements. Nevertheless, Arrays and Dictionarys share many
features in common, such as their access method (at: and at:put:), and
the ability to respond to collect:, select:, and many other messages.
The table below lists some of the characteristics of several forms

of collections:

Name	Creation Method	Size fixed?	Ordered?	Insertion method	Access method
Bag/Set	new	no	no	add:	includes:
Dictionary	new	no	no	at:put:	at:
Interval	n to: m	yes	yes	none	at:
List	new	no	yes	addFirst: addLast:	first last
Array	new:	yes	yes	at:put:	at:
String	new:	yes	yes	at:put:	at:

Examples:	Printed result:
i <- ’abacadabra’	
i size	10
i asArray	#($a $b $a $c $a $d $a $b $r $a)
i asBag	Bag ($a $a $a $a $a $r $b $b $c $d)
i asSet	Set ($a $r $b $c $d)
i occurrencesOf: $a	5
i reject: [:x | x isVowel]	bcdbr

The Collection class responds to the following methods:

addAll: aCollection

The argument must be a Collection. Add all the elements of the argument collection to the receiver collection.

asArray

Return a new collection of type Array containing the elements from the receiver collection. If the receiver was ordered, the elements will be in the same order in the new collection, otherwise the elements will be in an arbitrary order.

asBag

Return a new collection of type Bag containing the elements from the receiver collection.

asList

Return a new collection of type List containing the elements from the receiver collection. If the receiver was ordered, the elements will be in the same order in the new collection, otherwise the elements will be in an arbitrary order.

asSet

Return a new collection of type Set containing the elements
from the receiver collection.

asString

Return a new collection of type String containing the
elements from the receiver collection. The elements to be included
must all be of type Character. If the receiver was ordered,
the elements will be in the same order in the new collection, otherwise
the elements will be listed in an arbitrary order.

coerce: aCollection

The argument must be a Collection. Return a collection,
of the same type as the receiver, containing elements from the argument
collection. This message is redefined in most subclasses of
Collection.

collect: aBlock

The argument must be a one argument block. Return a new collection,
like the receiver, containing the result of evaluating the argument
block on each element of the receiver collection.

detect: aBlock

The argument must be a one argument block. Return the first element
in the receiver collection for which the argument block evaluates true.
Report an error and return "nil" if no such element exists. Note that
in unordered collections (such as Bags or Dictionarys)
the first element to be encountered that will satisfy the condition may
not be easily predictable.

detect: aBlock ifAbsent: exceptionBlock

Return the first element in the receiver collection for which the first
argument block evaluates true. Return the result of evaluating the
second argument if no such element exists.

includes: anObject

Return true if the receiver collection contains the argument.

inject: thisValue into: binaryBlock

The first argument must be a value, the second a two argument block.
The second argument is evaluated once for each element in the receiver
collection, passing as arguments the result of the previous evaluation
(starting with the first argument) and the element. The value returned
is the final value generated.

isEmpty

Return true if the receiver collection contains no elements.

occurrencesOf: anObject

Return the number of times the argument occurs in the receiver collection.

remove: oldObject

Remove the argument from the receiver collection. Report an error if the element is not contained in the receiver collection.

remove: oldObject ifAbsent: exceptionBlock

Remove the first argument from the receiver collection. Evaluate the second argument if not present.

reject: aBlock

The argument must be a one argument block. Return a new collection like the receiver containing all elements for which the argument block returns false.

select: aBlock

The argument must be a one argument block. Return a new collection like the receiver containing all elements for which the argument block returns true.

size

Return the number of elements in the receiver collection.

shallowCopy

Return a copy of the receiver.

printString

print the Collection into the Status Window.

1.36 Bags & Sets Classes:

Bags and Sets are each unordered collections of elements. Elements in the collections do not have keys, but are added and removed directly. The difference between a Bag and a Set is that each element can occur any number of times in a Bag, whereas only one copy is inserted into a Set.

Examples: Printed result:

i <- (1 to: 6) asBag Bag (1 2 3 4 5 6)

i size 6

i select: [:x | (x intNegRem: 2) strictlyPositive] Bag (1 3 5)

i collect: [:x | x intNegRem: 3] Bag (0 0 1 1 2 2)

j <- (i collect: [:x | x intNegRem: 3]) asSet Set (0 1 2)

j size 3

Note: Since Bags and Sets are unordered, there is no way to establish a mapping between the elements of the Bag i in the example above and the corresponding elements in the collection that

resulted from the message collect: [:x | x intNegRem: 3].

This Class responds to the following methods:

new

(Set only) Initialize a new instance of Set.

add: newElement

Add the indicated element to the receiver collection.

add: newObj withOccurences: anInteger

(Bag only) Add the indicated element to the

receiver Bag the given number of times.

first

Return the first element from the receiver collection. As the collection is unordered, the first element depends upon certain values in the internal representation, and is not guaranteed to be any specific element in the collection.

next

Return the next element in the collection. In conjunction with first, this can be used to access each element of the collection in turn.

remove: oldElement ifAbsent: exceptionBlock

Remove the element from a Bag or Set or evaluate the exceptionBlock if

the oldElement is NOT present.

size

Return the number of Elements in the Set or Bag.

occurrencesOf: anElement

ˆ dict at: anElement ifAbsent: [0] "for a Bag."

ˆ (list includes: anElement) ifTrue: [1] ifFalse: [0] "for a Set."

1.37 KeyedCollection Class:

The class KeyedCollection provides protocol for collections with keys, such as Dictionarys and Arrays. Since each entry in the collection has both a key and value, the method add: is no longer appropriate. Instead, the method at:put:, which provides both a key and a value, must be used.

Examples: Printed result:

i <- ’abacadabra’

i atAll: (1 to: 7 by: 2) put: $e ebecedebra

i indexOf: $r 9

i atAll: i keys put: $z zzzzzzzzzz

i keys Set (1 2 3 4 5 6 7 8 9 10)

i values Bag ($z $z $z $z $z $z $z $z $z $z)

#(how odd) asDictionary Dictionary (1 @ #how 2 @ odd)

This class responds to the following methods:

add: anElement

Returns an error String (no key!).

addAll: aCollection

Add the elements of the argument to the Receiver.

asDictionary

Return a new collection of type Dictionary containing the
elements from the receiver collection.

at: key

Return the item in the receiver collection whose key matches the
argument. Produces and error message, and returns nil, if no
item is currently in the receiver collection under the given key.

at:ifAbsent:

Return the element stored in the dictionary under the key given by the
first argument. Return the result of evaluating the second argument if
no such element exists.

atAll: aCollection put: anObject

The first argument must be a collection containing keys valid for the
receiver. At each location given by a key in the first argument
place the second argument.

binaryDo: aBlock

The argument must be a two argument block. This message is similar to
do:, however both the key and the element value are passed as
arguments to the block.

includesKey: key

Return true if the indicated key is valid for the receiver collection.

indexOf: anElement

Return the key value of the first element in the receiver collection
matching the argument. Produces an error message if no such element
exists. Note that, as with the message detect:, in unordered
collections the first element may not be related in any way to the
order in which elements were placed into the collection, but is rather
implementation dependent.

indexOf: anElement ifAbsent: exceptionBlock

Return the key value of the first element in the receiver collection
matching the argument. Return the result of evaluating the second

argument if no such element exists.

select: aBlock

Select elements from the Collection based on their values.

keys

Return a Set containing the keys for the receiver collection.

keysDo: aBlock

The argument must be a one argument block. Similar to do:, except that the values passed to the block are the keys of the receiver collection.

keysSelect: aBlock

Similar to select, except that the selection is made on the basis of keys instead of values.

remove: anElement

Returns an error String -- (no key!). (Default behavior).

removeKey: key

Remove the object with the given key from the receiver collection. Print an error message, and return nil, if no such object exists. Return the value of the deleted item.

removeKey: key ifAbsent: exceptionBlock

Remove the object with the given key from the receiver collection. Return the result of evaluating the second argument if no such object exists.

values

Return a Bag containing the values from the receiver collection.

1.38 Dictionary Class:

A Dictionary is an unordered collection of elements, as are Bags and Sets. However, unlike these collections, elements inserted and removed from a Dictionary must reference an explicit key. Both the key and value portions of an element can be any object, although commonly the keys are instances of Symbol or Number.

Examples: Printed result:

i <- Dictionary new

i at: #abc put: #def

i at: #pqr put: #tus

i at: #xyz put: #wrt

i print Dictionary (#abc @ #def #pqr @ #tus #xyz @ #wrt) i size 3

i at: #pqr #tus

i indexOf: #tus #pqr

i keys Set (#abc #pqr #xyz)

i values Bag (#wrt #def # tus)

This class responds to the following methods:

new

Initialize a new Dictionary, 17 elements in size.

hashNumber: aKey

Compute the hash Number for the given Key.

getList: aKey

Return a List starting at aKey.

at: aKey put: anObject

Place the second argument into the receiver under the key given by

the first argument.

removeKey: aKey ifAbsent: exceptionBlock

Remove an entry from the Dictionary.

findAssociation: aKey inList: linkedList

If aKey is in the linkedList, return the item, else return nil.

currentKey

Return the key of the last element yielded in response to a first

or next Method.

first

Return the first element of the receiver collection. Return nil

if the receiver collection is empty.

next

Return the next element of the receiver collection, or nil if

no such element exists.

printString

Display the currentKey & associated value as a Point.

checkBucket: bucketNumber

Check to see if the bucketNumber is nil, if it is, return nil,

otherwise return the first element of the currentList.

1.39 AmigaTalk Class:

The class AmigaTalk provides protocol for the pseudo-variable

amigatalk or smalltalk (use amigatalk in new code).

Since it is a subclass of Dictionary, this variable can be used to store information, and thus provide a means of communication between objects.

Other messages modify various parameters used by the AmigaTalk system.

This class is set up as a Singleton class, so that there is only one copy

of the global Dictionary.

ALL singleton classes contain the following:

the methods: isSingleton AND privateSetup AND

uniqueInstance Class instance variable.

Examples: Printed result:

atalk <- AmigaTalk new

atalk date Fri Apr 12 16:15:42 1985

atalk perform: #+ withArguments: #(2 5) 7

atalk doPrimitive: 10 withArguments: #(2 5) 7

AmigaTalk responds to the following methods:

isSingleton

Simply returns true.

getByteCodeArrayFrom: aClass for: aMethodString

Returns a ByteCodeArray from aClass for aMethodString, if aClass

responds to aMethodString. This method is only for examining

how a method has been translated into byteCodes.

date

Return the current date and time as a string.

printString

Override the parent printString method.

globalDictionary

Return the global Dictionary object.

addGlobal: newGlobal key: newKey

Add a new entry to the global Dictionary.

clearScreen

Erase any Curses or Plot3 windows.

debug: n

Change the AmigaTalk debug flag to n (0 = OFF or 1 = ON).

display

Set execution display (Status Window) to display the result of every

expression typed, but not for assignments. Note that the display

behavior can also be modified using the PRINTCMD=1 (formerly -d1)

argument on the command line.

There is also a menu item for this, attached to the Command Window.

displayAssign

Set execution display to display the result of every expression typed,

including assignment statements. Equivalent to using the PRINTCMD=2
argument when first starting the AmigaTalk system.
There is also a menu item for this, attached to the Command Window.
noDisplay
Turn off execution display - no results will be displayed unless
explicitly requested by the user.
There is also a menu item for this, attached to the Command Window.
doPrimitive: primNumber withArguments: argArray
Execute the indicated primitive with arguments given by the second
array. A few primitives (such as those dealing with process management) cannot be executed in this manner.
sh: sysCommand
The argument, which must be a String, is executed as an AmigaDOS
command by the shell. The value returned is the termination status
number of the shell.
WARNING: Know what you're doing when you use this method!
time: aBlock
The argument must be a block. The block is executed, and the number
of seconds elapsed during execution returned. Time is only accurate
to within about one second.
getProcessAddress: procName
Return an Integer representing the Address of the named Amiga-OS
Process.
getTaskAddress: taskName
Return an Integer representing the Address of the named Amiga-OS
Task.
getScreenAddress: screenName
Return an Integer representing the Address of the named Amiga-OS
Screen. screenName is the displayed title of the Screen .
getWindowAddress: windowName
Return an Integer representing the Address of the named Amiga-OS
Window. windowName is the displayed title of the Window .
getStringAddress: aString
Return an Integer representing the Address of aString.
NOT Kosher smalltalk, DO NOT USE!
getIntegerAddress: anInteger
Return an Integer representing the Address of anInteger.
NOT Kosher smalltalk, DO NOT USE!
getByteArrayAddress: aByteArray

Return an Integer representing the Address of aByteArray.

NOT Kosher smalltalk, DO NOT USE!

getTaskAddressList

Return an Array of Amiga-Task addresses.

getProcessAddressList

Return an Array of Amiga-Process addresses.

getScreenAddressList

Return an Array of Screen addresses.

getWindowAddressList

Return an Array of Window addresses.

showTaskProcessList

Display a Requester that lists all current System Tasks & Processes.

Returns an Integer representing the address of the last structure

selected in the ListView.

showScreenWindowList

Display a Requester that lists all current System Screens & Windows.

Returns an Integer representing the address of the last structure

selected in the ListView.

displayProcessInfo: procAddress

Display a Requester that lists the System Process structure.

displayTaskInfo: taskAddress

Display a Requester that lists the System Task structure.

displayScreenInfo: screenAddress

Display a Requester that lists the System Screen structure.

displayWindowInfo: windowAddress

Display a Requester that lists the System Window structure.

newIO: msgString title: title

Initialize the instance variables used for methods that allow the

User to use Amiga GUIs to get Strings, Integers, ScreenModes, display

Files, display Strings or to display Integers. This method is

equivalent to calling setIOMessage: followed by setIOTitle:

setIOMessage: newMessage

Change the display message for getString, getInteger, displayString &

displayInteger.

setIOTitle: newTitle

Change the display title for getString, getInteger, displayString &

displayInteger.

setIODirectory: newDirectory

Change the starting directory for getFileName. This method is identical to setIOMessage, but it's easier to see what your program is

doing if you call getFileName afterwards.

setIOScreenName: newScreenName

Change the Screen Name for getScreenModeID. This method is

identical to setIOMessage, but it's easier to see what your program is

doing if you call getScreenModeID afterwards.

getString

Show the User a GUI that asks them to enter a String .

NOTE: newIO:title: has to be called before this method, in

order to have a Requester title & a Request to display!

getInteger

Show the User a GUI that asks them to enter an Integer .

NOTE: newIO:title: has to be called before this method!

getFileName

Show the User the ASL file Requester & ask them to enter a filename.

NOTE: newIO:title: has to be called before this method!

getScreenModeID

Show the User the ASL ScreenMode Requester & ask them to select a

screen mode. This method DOES NOT change the current Screen Mode

being used, it simply returns an Integer that corresponds to the

ScreenModeID selected.

NOTE: newIO:title: has to be called before this method!

displayFile: fileName

Display the contents of a file to the User, using the contents of the

FileDisplayer ToolType as the file display program.

NOTE: newIO:title: has to be called before this method!

displayString: string

Display a String to the User in a GUI.

NOTE: newIO:title: has to be called before this method!

displayInteger: integer

Display an Integer to the User in a GUI.

NOTE: newIO:title: has to be called before this method!

listClassDictionaryTo: fileName indent: numSpaces

Write a list of all Classes currently known to AmigaTalk to the

given fileName, indenting subclasses by numSpaces.

listClassesOf: classObj to: fileName indent: numSpaces

Write a list of all subclasses of classObj currently known to

AmigaTalk to the given fileName, indenting subclasses by

numSpaces.

fileInPrimitiveFile: fileName

Read in a primitive file & incorporate it into the current AmigaTalk

environment. Primitive files end with .p (NOT enforced)

& represent parsed Class source code.

WARNING: Since there is no way of checking, make sure that the file

is debugged BEFORE you use this!

activeScreen

Returns a Screen Object that represents the currently active Screen.

activeWindow

Returns a Window Object that represents the currently active Window

or nil.

addUserScript: scriptMenuName toCall: scriptFileName

Adds the given scriptMenuName to the USER SCRIPTS menu of the main

AmigaTalk Command Window & allows the User to select & execute the

scriptFileName as if it were selected & loaded from Load Commands File...

Returns true if successful.

NOTE: Added menu items are currently NOT saved when the AmigaTalk program

exits. If you want a permanent menu item, add it to the

InitializeCommands Script that AmigaTalk executes on startup &

be sure to remove them using the removeUserScript: method in the

UpdateCommands Script. This is necessary to clean up the

memory allocations used by this method. These Scripts are both located

in the AmigaTalk:C/ directory.

removeUserScript: scriptMenuName

Removes the given scriptMenuName from the USER SCRIPTS menu of the main

AmigaTalk Command Window.

NOTE: All other methods are purposely NOT documented because the User

should not be using them.

1.40 SequenceableCollection Class:

The class SequenceableCollection contains protocol for collections

that have a definite sequential ordering and are indexed by integer

keys. Since there is a fixed order for elements, it is possible to

refer to the last element in a SequenceableCollection.

Examples: Printed result:

i <- ’abacadabra’

i copyFrom: 4 to: 8 cadab

i copyWith: $z abacadabraz

i copyWithout: $a bcdbr

i findFirst: [:x | x > $m] 9

i indexOfSubCollection: ’dab’ startingAt: 16

i reversed arbadacaba

i , i reversed abacadabraarbadacaba

i sort: [:x :y | x >= y] rdcbbaa

This class responds to the following methods:

, aCollection

Appends the argument collection to the receiver collection, returning a new collection of the same type as the receiver.

copyFrom: start to: stop

Return a new collection, like the receiver, containing the designated sub-portion of the receiver collection.

copyWith: newElement

Return a new collection, like the receiver, with the argument added to the end.

copyWithout: oldElement

Return a new collection, like the receiver, with all occurrences of the argument removed.

equals: aSubCollection startingAt: anIndex

The first argument must be a SequenceableCollection. Return true if each element of the receiver collection is equal to the corresponding element in the argument offset by the amount given in the second argument.

findFirst: aBlock

Find the key for the first element whose value satisfies the argument block. Produce an error message if no such element exists.

findFirst: aBlock ifAbsent: exceptionBlock

Both arguments must be blocks. Find the key for the first element whose value satisfies the first argument block. If no such element exists return the value of the second argument.

findLast: aBlock

Find the key for the last element whose value satisfies the argument block. Produce an error message if no such element exists.

findLast: aBlock ifAbsent: exceptionBlock

Both arguments must be blocks. Find the key for the last element whose value satisfies the first argument block. If no such element exists return the value of the second argument block.

firstKey

Return the first key valid for the receiver collection.

indexOfSubCollection: aSubColl startingAt: anIndex

Starting at the position given by the second argument, find the next block of elements in the receiver collection which match the collection given by the first argument, and return the index for the start of that block. Produce an error message if no such position exists.

indexOfSubCollection: aSubColl startingAt: anIndex ifAbsent: exceptBlk

Similar to indexOfSubCollection:startingAt:, except that the result of the exception block is produced if no position exists matching the pattern.

last

Return the last element in the receiver collection.

lastKey

Return the last key valid for the receiver collection.

replaceFrom: start to: stop with: replacementCollection

Replace the elements in the receiver collection in the positions indicated by the first two arguments with values taken from the collection given by the third argument.

replaceFrom: first to: stop with: repColl startingAt: repStart

Replace the elements in the receiver collection in the positions indicated by the first two arguments with values taken from the collection given in the third argument, starting at the position given by the fourth argument.

reversed

Return a collection, like the receiver, with elements reversed.

reverseDo: aBlock

Similar to do:, except that the items are presented in reverse order.

select: aBlock

Return a new Collection like the receiver containing all elements for which the argument Block returns true.

sort

Return a collection, like the receiver, with the elements sorted using the comparison <=. Elements must be able to respond to the binary message <=.

sort: sortBlock

The argument must be a two argument block which yields a boolean. Return a collection, like the receiver, sorted using the argument to compare elements for the purpose of ordering.

with: aSequencableCollection do: aBlock
The second argument must be a two argument block. Present one element from the receiver collection and from the collection given by the first argument in turn to the second argument block. An error message is given if the collections do not have the same number of elements.

1.41 Interval Class:

The class Interval represents a sequence of numbers in an arithmetic sequence, either ascending or descending. Instances of Interval are created by Numbers in response to the message to: or to:by:. In conjunction with the message do:, Intervals create a control structure similar to do or for loops in Algol-like languages. For example:
(from: 1 to: 10 by: 2) do: [:x | x print]
will print the even numbers from 2 to 10. Although they are a collection, Intervals cannot be added to. They can, however, be accessed randomly using the message at:ifAbsent:.
Examples: Printed result:
(7 to: 13 by: 3) asArray #(7 10 13)
(7 to: 13 by: 3) at: 2 10
(1 to: 10) inject: 0 into: [:x :y | x + y] 55
(7 to: 13) copyFrom: 2 to: 5 #(8 9 10 11)
(3 to: 5) copyWith: 13 #(3 4 5 13)
(3 to: 5) copyWithout: 4 #(3 5)
(2 to: 4) equals: (1 to: 4) startingAt: 2 True
This class responds to the following methods:
first
Produce the first element from the interval. Note that Intervals also respond to the message at:ifAbsent:, which can be used to produce elements in an arbitrary order.
last
Produce the last element from the interval. Note that Intervals also respond to the message at:ifAbsent:, which can be used to produce elements in an arbitrary order.
from: lowerBound to: upperBound by: stepSize
Initialize the upper and lower bounds and the step size for the receiver. (This is also used internally by methods in Number to create new Intervals).
next

Produce the next element from the Interval.

size

Return the number of elements that will be generated in producing the interval.

inRange: value

Return true if value is within the Interval boundaries.

at: index ifAbsent: exceptionBlock

If the value lies within the Interval boundaries, return the value, else evaluate the exceptionBlock.

printString

Display the Interval in the Status Window.

coerce: newCollection

Transform the Interval into an Array.

at: index put: value

This method is NOT valid for Intervals & returns an error String.

add: val

This method is NOT valid for Intervals & returns an error String.

removeKey: key ifAbsent: exceptionBlock

This method is NOT valid for Intervals & returns an error String.

deepCopy

Return a copy of the Interval.

shallowCopy

Same as deepCopy method.

1.42 LinkedList Class:

Lists represent collections with a fixed order, but indefinite size. No keys are used, and elements are added or removed from one end of the other. Used in this way, Lists can perform as stacks or as queues. The table below illustrates how stack and queue operations can be implemented in terms of messages to instances of List.

Examples: Printed result:

i <- List new

i addFirst: 2 / 3 List (0.6666)

i add: $A

i addAllLast: (12 to: 14 by: 2)

i print List (0.6666 $A 12 14)

i first 0.6666

i removeLast 14

i print List (0.6666 $A 12)

stack operations queue operations

---

push addLast: add addLast:

pop removeLast first in queue first

top last remove first in queue removeFirst

test empty isEmpty test empty isEmpty

This class responds to the following methods:

add: anItem

Add the element to the beginning of the receiver collection. This is the same as addFirst:.

addAllFirst: aCollection

The argument must be a SequenceableCollection. The elements of the argument are added, in order, to the front of the receiver collection.

addAllLast: aCollection

The argument must be a SequenceableCollection. The elements of the argument are added, in order, to the end of the receiver collection.

addFirst: anItem

The argument is added to the front of the receiver collection.

addLast: anItem

The argument is added to the back of the receiver collection.

remove: anItem

Remove the given element from the List.

remove: anItem ifAbsent: exceptionBlock

Remove an element from the List if it's present. If it's absent, evaluate the exceptionBlock.

removeFirst

Remove the first element from the receiver collection, returning the removed value.

removeLast

Remove the last element from the receiver collection, returning the removed value.

first

Return the first element in the List.

next

Return the next element in the List.

current

Return the current element in the List.

last

Return the last element in the List.

isEmpty

Return true if the List is empty, false otherwise.

removeError

Return a string indicating that the User cannot remove from an empty List.

coerce: aCollection

Transform aCollection into a List Object.

1.43 Semaphore Class:

Semaphores are used to synchronize concurrently running Processes. This class is NOT the same as the Semaphores used by the AmigaOS. This class responds to the following methods:

new

A Semaphore starts out with zero excess signals when created by this method.

new: numberOfSignals

A Semaphore can be created with an arbitrary number of excess signals with this method.

signal

If there is a process blocked on the semaphore it is scheduled for execution, otherwise the number of excess signals is incremented by 1.

wait

If there are excess signals associated with the semaphore the number of signals is decremented by one, otherwise the current process is placed on the semaphore queue.

1.44 File Class:

A File is a type of collection where the elements of the collection are stored on an external medium, typically a disk. For this reason, although most operations on collections are defined for files, many can be quite slow in execution. A file can be opened in one of three modes: In character mode every read returns a single character from the file. In integer mode every read returns a single word, as an integer value. In string mode every read returns a single line, as a

String. For writing, character and string modes will write the string representation of the argument, while integer mode must write only a single integer.

Responds To

at: aPosition

Return the object stored at the indicated position. Position is given as a character count from the start of the file.

at: aPosition put: anObject

Place the object at the indicated position in the file. Position is given as a character count from the start of the file.

modeCharacter

Set the mode of the receiver file to character.

currentKey

Return the current position in the file, as a character count from the start of the file.

modeInteger

Set the mode of the receiver file to integer.

open: aName

Open the indicated file for reading. The argument must be a String.

open: aName for: opType

The for: argument must be one of r, w or r+ (see fopen(3) in the Unix programmers manual). Open the file in the indicated mode.

close

Close a previously opened file.

read

Return the next object from the file.

size

Return the size of the file, in character counts.

modeString

Set the mode of the receiver file to string.

write: anObject

Write the argument into the file.

1.45 ArrayedCollection Class:

The class ArrayedCollection provides protocol for collections with a fixed size and integer keys. Unlike other collections, which are created using the message new, instances of ArrayedCollection must be created using the one argument message new:. The argument given with this message must be a positive integer, representing the size of the collection to be created. In addition to the protocol shown, many of the methods inherited from superclasses are redefined in this class.

Examples: Printed result:

’small’ = ’small’ True

’small’ = ’SMALL’ False

’small’ asArray #($s $m $a $l $l)

’small’ asArray = ’small’ True

#(1 2 3) padTo: 5 #(1 2 3 nil nil)

#(1 2 3) padTo: 2 #(1 2 3)

This class responds to the following methods:

= anArray

The argument must also be an Array. Test whether the receiver and the argument have equal elements listed in the same order.

at: key ifAbsent: exceptionBlock

Return the element stored with the given key. Return the result of evaluating the second argument if the key is not valid for the receiver collection.

coerce: aCollection

Transform aCollection to an ArrayedCollection.

copyFrom: start to: stop

Return a new portion of the ArrayedCollection.

currentKey

Return the current key value.

deepCopy

Return a copy of the ArrayedCollection. This method differs from shallowCopy in that more memory space is allocated from the system.

do: aBlock

Perform aBlock for each element of the ArrayedCollection.

first

Return the first element of the ArrayedCollection.

firstKey

Return the index of the first element (which is always one).

lastKey

Return the index of the last element (which is equal to the size).

next

Return the next element of the ArrayedCollection.

padTo: length

Return an array like the received that is at least as long as the
argument value. Returns the receiver if it is already longer than the
argument.
shallowCopy
Return a copy of the ArrayedCollection.

1.46 Array Class:

Instances of the class Array are perhaps the most commonly used data
structure in Smalltalk programs. Arrays are represented textually by
a pound sign preceding the list of array elements.
Examples: Printed result:
i <- #(110 101 97)
i size 3
i <- i grow: 116 #(110 101 97 116)
i <- i collect: [:x | x asCharacter] #(#n #e #a #t)
i asString neat
This class responds to the following methods:
at: index
Return the item stored in the position given by the argument. An error
message is produced, and nil returned, if the argument is not
a valid key.
at: index put: value
Store the second argument in the position given by the first argument.
An error message is produced, and nil returned, if the
argument is not a valid key.
grow: newElement
Return a new array one element larger than the receiver, with the
argument value attached to the end. This is a slightly more efficient
command than copyWith:, although the effect is the same.
printString
Display the elements of the Array in the Status Window.
size
Return the number of elements in the Array.
new: newSize
Return a new instance of Array of the given size.

1.47 ByteArray Class:

A ByteArray is a special form of array in which the elements must be numbers in the range 0-255. Instances of ByteArray are given a very compact encoding, and are used extensively internally in the AmigaTalk system. A ByteArray can be represented textually by a pound sign preceding the list of array elements surrounded by a pair of square braces.

Examples: Printed result:

i <- #[110 101 97]

i size 3

i <- i copyWith: 116 #[110 101 97 116]

i <- i asArray collect: [:x | x asCharacter] #(#n #e #a #t)

i asString neat

This class responds to the following methods:

at: index

Return the item stored in the position given by the argument. An error message is produced, and nil returned, if the argument is not a valid key.

at: index put: value

Store the second argument in the position given by the first argument. An error message is produced, and nil returned, if the argument is not a valid key.

printString

Display a representation of the array in the status window.

displayBytes: title

Display the array in a Requester with the given title. This method is substantially faster than printString for large ByteArrays.

size

Return the number of elements in the array.

new: numElements

Make a new instance of the ByteArray Class as large as the given size. The elements are initialized to zero.

1.48 String Class:

Instances of the class String are similar to Arrays, except that the individual elements must be Character. Strings are represented literally by placing single quote marks around the characters making up the string.

Strings also differ from Arrays in that Strings possess an ordering,

given by the underlying ASCII sequence.

Examples: Printed result:

’example’ at: 2 $x

’bead’ at: 1 put: $r read

’small’ > ’BIG’ True

’small’ sameAs: ’SMALL’ True

’tary’ sort arty

’Rats live on no evil Star’ reversed ratS live on no evil staR

This class responds to the following methods:

, aString

Concatenates the argument to the receiver string, producing a new

String. If the argument is not a String it is first converted

using printString.

= aString

Return true if the Receiver is the same as the arugment.

< aString

The argument must be a String. Test if the receiver is

lexically less than the argument. For the purposes of

comparison, case differences are ignored.

<= aString

Test if the receiver is lexically less than or equal to the argument.

>= aString

Test if the receiver is lexically greater than or equal to the argument.

> aString

Test if the receiver is lexically greater than the argument.

cr

Return newline (ASCII value 10) as a String object.

asSymbol

Return a Symbol with characters given by the receiver string.

at: aNumber

Return the character stored at the position given by the argument.

Produce an error message, and return nil, if the argument

does not represent a valid key.

at: aNumber put: aChar

Store the character given by second argument at the location given by

the first argument. Produce an error message, and return nil,

if either argument is invalid.

compareError

Return an error String about string comparison.

copyFrom: start length: len

Return a substring of the receiver. The substring is taken from the indicated starting position in the receiver and extends for the given length. Produce an error message, and return nil, if the given positions are not legal.

copyFrom: start to: stop

Return a substring of the receiver. The substring is taken from the indicated positions. Produce an error message, and return nil, if the given positions are not legal.

deepCopy

Return a copy of the Receiver.

new: size

Make a new String filled with blanks of the size given.

NOTE: the maximum string length is silently limited to 512 characters.

printAt: aPoint

The argument must be a Point which describes a location on the Curses screen. The string is printed at the specified location.

printString

Print the Receiver (with surrounding quote marks) on the Status Window.

print

Print the Receiver (with NO surrounding quote marks) on the Status Window.

size

Return the number of characters stored in the string.

sameAs: aString

Return true if the receiver and argument string match with the exception of case differences. Note that the boolean message =, inherited from ArrayedCollection, can be used to see if two strings are the same including case differences.

1.49 Block Class:

Although it is easy for the programmer to think of blocks as a syntactic construct, or a control structure, they are actually objects, and share attributes of all other objects in the Smalltalk system, such as the ability to respond to messages.

Examples: Printed result:

['block indeed'] value block indeed

[:x :y | x + y + 3] value: 5 value: 7 15

This Class responds to the following methods:

fork

Start the block executing as a Process. The value nil is

immediately returned, and the Process created from the

block is scheduled to run in parallel with the current process.

forkWith: argumentArray

Similar to fork, except that the array is passed as

arguments to the receiver block prior to scheduling for execution.

newProcess

A new Process is created for the block, but is not scheduled

for execution.

newProcessWith: argumentArray

Similar to newProcess, except that the array is passed

as arguments to the receiver block prior to it being made into a

process.

value

Evaluates the receiver block. Produces an error message, and returns

nil, if the receiver block required arguments.

Return the value yielded by the block.

value: a

Evaluates the receiver block. Produces an error message, and returns

nil, if the receiver block did not require a single argument.

Return the value yielded by the block.

value: a value: b

Two argument block evaluation.

value:a value: b value: c

Three argument block evaluation.

value: a value: b value: c value: d

Four argument block evaluation.

value: a value: b value: c value: d value: e

Five argument block evaluation.

whileTrue: aBlock

The receiver block is repeatedly evaluated. While it evaluates to

true, the argument block is also evaluated. Return nil when

the receiver block no longer evaluates to true.

whileTrue

The receiver block is repeatedly evaluated until it returns a value
that is not true.
whileFalse: aBlock
The receiver block is repeatedly evaluated. While it evaluates to
false, the argument block is also evaluated. Return nil when
the receiver block no longer evaluates to false.
whileFalse
The receiver block is repeatedly evaluated until it returns a value
that is not false.

1.50 Class Class:

The class Class provides protocol for manipulating class instances. An
instance of class Class is generated for each class in the AmigaTalk
system. New instances of this class are then formed by sending messages
to the class instance.
Examples: Printed result:
Array new: 3 #(nil nil nil)
Bag respondsTo: #add: True
SequenceableCollection superClass KeyedCollection
The methods that Class responds to are:
edit
The user is placed into a editor editing the file from which the class
description was originally obtained. When the editor terminates, the
class description will be re-parsed and will override the previous
description. See also view.
list
Lists all subclasses of the given class recursively. In particular,
Object list will list the names of all the
classes in the system.
new
A new instance of the receiver class is returned. If the methods for
the receiver contain protocol for new, the new instance will
first be passed this message before returning.
new: aValue
A new instance of the receiver class is returned. If the methods for
the receiver contain protocol for new:, the new instance will
first be passed this message before returning.
printClassString

Return a Symbol representing the argument Class.

respondsTo

List all the messages that the current class will respond to.

respondsTo: aSymbol

The argument must be a Symbol. Return true if the receiver class, or any of its superclasses, contains a method for the indicated message. Return false otherwise.

superClass

Return the superclass of the receiver class.

variables

Return an array containing the names of the instance variables used in the receiver class.

view

Place the user into an editor viewing the class description from which the class was created. Changes made to the file will not, however, affect the current class representation.

getByteArray: methodString

Return a ByteArray that represents the given method in the Receiver.

1.51 Process Class:

Processes are created by the system, or by passing the message newProcess or fork to a block; they cannot be created directly by the user.

This Class responds to the following methods:

block

The receiver process is marked as being blocked. This is usually the result of a Semaphore wait. Blocked processes are not executed.

resume

If the receiver process has been suspended, it is rescheduled for execution.

suspend

If the receiver process is scheduled for execution, it is marked as suspended. Suspended processes are not executed.

state

The current state of the receiver process is returned as a Symbol.

termErr: msgName

Print a String describing action taken on a terminated Process.

terminate

The receiver process is terminated. Unlike a blocked or suspended process, a terminated process cannot be restarted.

unblock

If the receiver process is currently blocked, it is scheduled for execution.

yield

Returns nil. As a side effect, however, if there are pending processes, the current process is placed back on the process queue and another process started.