Smalltalk Frequently Asked Questions

Vikas Malik
vmalik@dc.infi.net
Knowledge Systems Corporation

May 14,1996

Contents

1 General Questions about Smalltalk

1.1 What is `self`?

`self` refers to the receiver of the message. It is usually used within a method to send additional messages to the receiver. `self` is frequently used when it is desired to pass the sender object (`self`), as a message argument, to a receiver who requires knowledege of the sender or who will in some way manipulate the sender.

1.2 What is `super`?

`super` refers to the superclass of the class that defines the message sent to the receiver. `super` is mechanism by which a sender can overide its own defined method hierarchy.

1.3 What are three different message types in Smalltalk?

1. Unary messages
2. Binary Messages
3. Keyword Messages

1.4 What is `shallowCopy`?

`shallowCopy` returns a copy of receiver which shares the receiver's instance variables.

1.5 What is deep copy?

`deepCopy` returns a copy of the receiver with shallow copies of each instance variable.

1.6 What are class instance variables?

Class instance variables are similar to class variables, except that they are created for each subclass of the defining class. When a class declares a class instance variable, a new variable is created for each subclass of that class. Each subclass then has its own instance of the variable and retains its own value for the variable, but each subclass has a variable with the same name. Only class methods of a class and its subclasses can refer to class instance variables; instance methods cannot.

1.7 What do you get when you inspect `Smalltalk`?

An instance of `SystemDictionary`. `SystemDictionary` maintains all the names of classes, global variables and pool dictionaries in the system. `Smalltalk` is the sole instance of the class `SystemDictionary`.

1.8 How can I implement stack and queue operations in Smalltalk?

An `OrderedCollection` can be used to implement stack and queue operations. A stack is implemented by using `addLast: anObject` and `removeLast` methods for pushing and poping respectively. A Queue is implemented by using `addLast: anObject` and `removeFirst` methods.

1.9 What is the difference between an array and a set?

An array is a fixed size collection whereas a set can grow in size. An array has integer keys and a set is not keyed. A set enforces uniqueness upon its members.

1.10 What is the difference between protocol and category?

The methods of a class are organised in logical groups called protocols. Classes are grouped together into groups called categories.

1.11 What do I get from `1 + 2 * 3`?

9

1.12 How do I stop the execution of a program?

By sending message `halt`.

1.13 What is the difference between = and == ?

= is a test for equality. == is a test for identity (same object).

Except for some special classes like Symbol, Character, and SmallInteger, two cases of a value (eg. 1.54) will not be the identical object, just one with equal value. The special cases are "special" and any variables that points to such an object will be by definition "Identical".

1.14 How do I iterate a collection?

By using the following enumeration messages.

1. `do: aBlock`

2. `select: aBlock`
3. `reject: aBlock`
4. `detect: aBlock`
5. `collect: aBlock`
6. `inject: aValue into: aBlock`

1.15 How do I use message `perform:` ?

`perform:` method is used to tell an object to execute a method whose name, rather than being hardcoded, is sent as a parameter. This means that the name of the message need not be know until runtime.

```
anObject perform: #methodName
```

1.16 What is the difference between `detect:` and `select:` methods for collections?

`detect: aBlock` returns the first element of the collection for which `aBlock` evaluates to `true`. `select: aBlock` returns a collection whose elements are the elements of the receiver for which `aBlock` evaluates to `true`.

1.17 What is the result of sending `collect: [ :each | each > 5 ]` to `aCollection`?

A collection whose elements are `true` and `false`.

`collect:aBlock` returns a collection which contains the results of performing the same operation on each element in the receiver collection.

1.18 What do I use to concatenate strings efficiently?

Streams.

1.19 What is superclass of Object?

`nil`.

1.20 What is `subclassResposiblity` method used for?

`subclassResponsibility` method is used to defer the actual implementation of a method.

1.21 What is `shouldNotImplement` method used for?

`shouldNotImplement` method is used to indicate that a subclass wants to undefine a method defined in a superclass.

1.22 What methods do I need to implement if I implement method for an object?

`hash:`

1.23 What method do I need to implement for proxy objects?

`doesNotUnderstand: aMessage`

1.24 What are three pseudo variables used by VisualWorks?

`self`, `super`, and `thisContext`.

1.25 What is the difference between `SmallInteger` and `LargeInteger`?

A `SmallInteger` has fixed number of bytes and a `LargeInteger` has variable number of bytes.

1.26 What is the difference between `Symbol` and `String`?

`Symbols` are similar to strings except that they are unique. This means that whilst one can create several string objects containing the same sequence of characters, there will only be exactly one instance of a symbol with a given sequence of characters.

1.27 What is the difference between `Bag` and `Set`?

A `Bag` can have duplicates whereas a `Set` contains no duplicate objects.

1.28 What is the difference between chaining and cascading?

In Chaining, one message can follow on after another. In this case the second message is sent to the object which is the result of the first message.

```
anObject msg1 msg2 msg3
```

In Cascading, each message is followed by a semicolon (;) and another message. In Cascading, subsequent messages are sent to the first receiver.

```
anObject msg1; msg2; msg3
```

1.29 What is `yourself` method used for?

`yourself` method returns `self`. It is usually used while cascading messages like `add:`.

```
| aCollection |

aCollection :=3D OrderedCollection new add: 1;
                                       add: 2;
                                       yourself.
```

In the above example `add:` returns its argument, not the receiver.

Therefore, `yourself` message is sent to do the proper assignment of an `OrderedCollection` to `aCollection` variable.

1.30 What is the difference between `isMemberOf:` and `isKindOf:` methods?

`isMemberOf: aClass` returns `true` if receiver is an instance of `aClass`.

`isKindOf: aClass` returns `true` if receiver is an instance of `aClass` or one of its subclasses.

1.31 Where is the method `new` defined?

Method `new` is defined on instance side of class `Behavior`.

1.32 Why is an `IdentitySet` faster than a `Set`?

An `IdentitySet` is faster that a `Set` because == is a faster test than =

1.33 What happens when I modify a collection while iterating over it?

Modifying a collection while iterating over it will give unpredictable results.

```
Employees do: [:anEmployee | anEmployee isProgrammer ifTrue: [ Employees

remove: anEmployee]]
```

Above example will not work as expected since we are changing the size of Employees collection while iterating over it. Making a copy of the collection will avoid the above problem.

```
Employees copy do: [:anEmployee | anEmployee isProgrammer ifTrue: [

Employees remove: anEmployee]]
```

1.34 What is `nil`?

`nil` is the only instance of the class `UndefinedObject` and it is the default value for any new variable until a specific value is assigned to it.

1.35 What are pool dictionaries?

Pool dictionaries are created for providing access to pool variables to several classes that are not related by inheritance.

1.36 What is a literal?

A literal is a piece of Smalltalk code that the compiler converts immediately to an object. Literals can be freely included in the programs just by typing them.

Literal type	Example
Character	$s
Symbol	#name
String	'David Letterman'
Number	12
Boolean	false
UndefinedObject	nil
Array of literals	#('apple' 12.9 name)

2 MVC

2.1 What is MVC?

MVC stands for Model-View-Controller. Model stores information/data. Model notifies its dependents whenever one or more of its variables is changed. View is the component for displaying output. Contoller is the component that enables the user to interact with, or control, the application.

2.2 What is an aspect?

An aspect is a piece or subset of model's domain information.

2.3 What is a dependent?

A dependent is an object that is dependent on the information residing in a model. A dependent object is usually a view, a window, or another model and is contained in dependents collection of a model.

2.4 How do I add a dependent to a model?

`aModel addDependent:`

`aDependent.`

This method adds `aDependent` to `aModel`'s dependents collection.

2.5 How do I release dependents from a model?

`aModel` release replaces `aModel`'s dependents collection with `nil`.

2.6 What is the changed/update mechanism?

The changed/update mechanism is used by a model to broadcast a notification of change to all its dependents.

2.7 What happens when I send changed or changed: message to a Model?

A changed/changed: message sends an update message to all of receiver's(model) dependents.

2.8 How do I implement an update method for a view?

The prototype for an update method for a view is

```
update: anAspect

      anAspect =3D anAspectOfInterest

       ifTrue: [ self invalidate]
```

2.9 What does invalidateRectangle: method do?

Invalidate messages initiate a view's redrawing process.

`invalidateRectangle: aRectangle` invalidates only that area of the view defined by `aRectangle`.

2.10 How does a controller accesses keyboard events and mouse state?

By using its instance variable sensor, which references a `TranslatingSensor`.

2.11 What is a `ValueHolder` in VisualWorks?

A `ValueHolder` is a value model. It holds simple model objects like numbers, strings, etc. A ValueHolder is created by sending `asValue` message to any object or sending `with: anObject` to `ValueHolder` class

```
anObject asValue.

ValueHolder with: anObject
```

A ValueHolder understands value/value: protocol. value message is used to access its value. value: anObject message is used to set its value to anObject.

2.12 What is a channel?

A channel is a value model that is used as a common access point for a changing value.

2.13 How do an object register interest in a ValueModel?

By sending `onChangeSend: aChangeMessage to: anInterestedObject` to a `ValueModel`. The advantages of this approach are:

- The interested object does not need to be a dependent of the value model.
- The interested object does not need to implement an update method.

2.14 What is a dependency transformer?

A `DependencyTransformer` implements the behavior needed by an object to register interest in a value model. A `dependencyTransformer` is defined as a dependent of a value model and converts an update message sent to itself into a specific change message and sends this change message to the interested object.

2.15 How do I change the value of a `ValueModel` without triggering any updates?

1. By using `setValue: newValue` method instead of `value: newValue` method. `setValue: method` replaces the value instance variable without sending update messages to dependents.
2. Remove the `DependencyTransformer` from the `ValueModel`'s dependents. Send `retractInterestsFor: anObject` message to the `ValueModel`. This is done just prior to sending `value: newValue` message.

Using `setValue: disallows` all updates. `retractInterestsFor:` only disallows a specific update, allowing all others to proceed.

2.16 What is an `AspectAdaptor`?

An `AspectAdaptor` is a `ValueModel` whose value actually belongs to another object called the subject. The task of `AspectAdaptor` is to interface the general-purpose view object to just one aspect of the model(subject). An `AspectAdaptor` is created as follows:

```
| aa |
      aa := AspectAdaptor subject: Employee new.
      aa forAspect: #ssn.
```

One can change the ssn of this employee by using `aa value: aNumber`. An `AspectAdaptor` needs to know which messages to send to the model to access and assign value to one of its aspect. In the above example value/value: messages sent to aa are converted into ssn/ssn: and sent to subject. `forAspect: #ssn` message sets the `getSelector` to `#ssn` and sets the `putSelector` to `#ssn:` The value method of `AspectAdaptor` is implemented as

```
value
          ^subject perform: self getSelector
```

2.17 What is the subject channel of an `AspectAdaptor`?

The subject channel is the `ValueHolder` containing the subject of an `AspectAdaptor`. It is useful when several `AspectAdaptor`s share the same subject and its value of this subject needs to be changed.

2.18 What is a `PluggableAdaptor`?

A `PluggableAdaptor` uses blocks to adapt a model to a view instead of using just selectors like `AspectAdaptor`. The blocks are called the `getBlock`, the `putBlock` and the `updateBlock`. These blocks get executed when a `PluggableAdaptor` receives value/value: and update messages. A `PluggableAdaptor` can be created as follows:

```
| pa |
      pa := PluggableAdaptor on: Employee new.
      pa getBlock: [ :m | m salary * 30 ]
         putBlock: [ :m :v | m salary: (v/30)]
         updateBlock: [ :m :a :p | ....  ].
```

On sending value message to `pa`, `getBlock` gets executed. It send salary message to employee(model), multiplies the result by 30 and returns it. In this way, one can perform more complex operations on model as a result of just sending the value message to `pluggableAdapator`.

2.19 What is an `IndexedAdaptor`?

An `IndexedAdaptor` is similar to `AspectAdaptor` except that its subject is a sequenceable collection and `value/value:` messages are dispatched to the subject as `at:/at: put:`.

2.20 What is a SelectionInList?

A `SelectionIinList` is a selection model. It has two instance variables.

1. `listHolder` instance variable is a ValueModel containing a sequenceable collection. `listHolder` acts as a model for `SequenceView`.
2. `selectionIndexHolder` instance variable is a `ValueHolder` with the index of the current selection as its value.

`SelectionInList` does not have any dependents. Both of its instance variables have two dependents: the `SequenceView` and the `SelectionInList` object itself.

2.21 What is a `BufferedValueHolder`?

A `BufferedValueHolder` is a `ValueModel` that references two other value models called its subject and trigger channel. When a `BufferedValueHolder` receives a value: message, it holds onto the new value and does not update the subject until trigger channel becomes true.

2.22 What is the difference between an `IndexedAdaptor` and an `AspectAdaptor`?

An `IndexedAdaptor` operates on a numbered instance variable in the subject, whereas an `AspectAdaptor` operates on a named instance variable.

3 Other VisualWorks Questions

3.1 What is a `BlockClosure` in VisualWorks?

Class `BlockClosure` implements the block notation in VisualWorks Smalltalk. In VisualWorks, blocks are close to being closures. One can declare variables local to the block, and the names of block parameters are local to the block.

3.2 What is the launcher in VisualWorks?

The launcher is the root of the VisualWorks development environment. Various development tools can be launched (or opened) from its menu. The launcher is implemented by `VisualLauncher` which is a subclass of `ApplicationModel`.

3.3 What is a SpecWrapper?

A VisualWorks component is a `SpecWrapper`. A `SpecWrapper` is wrapper that contains a widget, decoration for the widget, a copy of `WidgetState` object and `ComponentSpec`.

3.4 What is the difference between an active component and a passive component?

An active component is a VisualWorks component whose widget is a View and has a Model and a Controller. A passive component is a VisualWorks component whose widget is not a View and it does not depend on a model and a controller.

3.5 What is a widget?

A widget is a visual part responsible for the visual representation of a VisualWorks component.

3.6 What is difference between application model and aspect model?

An `ApplicationModel` is responsible for creating and managing a runtime user interface. An aspect model contains a single aspect of info and provides the model behavior for a single VisualWorks component. The relationship between an application model and an aspect model is that an application model contains one or more aspect models.

3.7 What is a keyboard hook?

A keyboard hook is a used for intercepting all the keyboard activity going to a VisualWorks component. It is a block which is evaluated just prior to the widget controller handling a keyboard event. Keyboard hook can be set as follows:

```
| comp |

comp := anApplicationModel builder componentAt:=
\#myComponent.

comp widget controller keyboardHook: aBlock.
```

3.8 What is the Transcript?

The `Transcript` is a text window used by the system to report important events. The global variable Transcript is an instance of the class TextCollector in VisualWorks and class `TranscriptWindow` in Digitalk.

3.9 What is a `dispatcher`?

A `dispatcher` is used by a widget controller to dispatch notification and validation messages to the `ApplicationModel`. A dispatcher is an instance of class `UIDispatcher`.

3.10 What is a `ComponentSpec`?

A `ComponentSpec` describes properties and features of a VisualWorks component. `ComponentSpec` provides behavior for interface persistence in form of source code or text file.

3.11 What is a builder in `ApplicationModel` architecture?

A builder is an instance of class `UIBuilder`. It is used to construct a user interface according to the specifications. It also provides access to the runtime interface. Builder provides access to named components, the keyboard processor, the window and aspect models of a running application.

3.12 What is a `FullSpec`?

A `FullSpec` is a combination of a window spec and a spec collection. A window spec describes a window and a spec collection is a collection of component specs for all components of a window.

3.13 What is a builder's resource?

A resource is an object used by the builder to construct the interface according to the specs. Examples of resources are aspect models, menus, images, and labels. There are two types of resources:

1. Static resources - do not change during runtime.
2. Dynamic resources - change during runtime.

3.14 What is the builder's source?

The builder's source is an `ApplicationModel` that provides the necessary resources to a builder for building an interface according to the specs. It is kept in source instance variable of `UIBuilder`.

3.15 How does a builder caches resources?

A builder uses the following variables to cache resources.

- bindings to cache aspect models reqired by active components.
- labels to cache text labels.
- visuals to cache visual components.

3.16 What is a `lookPolicy` object?

A `lookPolicy` object is an instance of one of the subclasses of the `UILookPolicy` abstract class. A `lookPolicy` object is used to create a VisualWorks component based on the `componentSpec` and the specific window environment's look and feel.

3.17 How can I change a builder's look policy?

By sending `policy: aLookPolicy` message to a builder.

```
aBuilder policy: MotifLookPolicy new.
```

3.18 How is a VisualWorks component built?

A VisualWorks component is built by either the component spec or by the builder's look policy object. Usually it is the look policy object that builds a VisualWorks component. A look policy object uses its instance methods, component building methods, to construct a VisualWorks component based on a `ComponentSpec` and look and feel of that particular look policy.

3.19 What methods are used to access window, `keyboardProcessor`, named components and aspect models from a builder?

```
aBuilder window.

aBuilder keyboardProcessor.

aBuilder componentAt: \#componentID.

aBuilder aspectAt: \#aspect.
```

3.20 What is a keyboard processor?

A keyboard processor is an instance of class `KeyboardProcessor`. It directs keyboard events to the current widget controller. There is only one keyboard processor per window. A widget controller that takes focus has an instance variable, `keyboardProcessor`, used to reference its window's keyboard processor.

3.21 What is the difference between `preBuildWith:` and `postBuildWith:` methods of `ApplicationModel`?

`preBuildWith: aBuilder` method allows `anApplicationModel` to make any changes to its builder prior to handling its full spec.

`postBuildWith: aBuilder` method allows any final changes to the interface prior to opening.

3.22 What is the difference between `postBuildWith:` and `postOpenWith:` methods of `ApplicationModel`?

In the `postBuildWith:` method, the interface is completely built but it exists in memory only and window is not open. Whereas `postOpenWith:` method allows the application model to make any changes to the interface with the window open.

3.23 How do I close a window programmatically?

By sending the `closeRequest` message to `anApplicationModel`.

`closeRequest` method sends an `update: #closeRequest` to `anApplication` Model dependents (window is one of the dependents).

3.24 Which message is sent to `anApplicationModel` for notification of a window close?

`noticeOfWindowClose: aWindow`.

3.25 Which message is sent to `anApplicationModel` for validation of a window close?

`requestForWindowClose`.

3.26 What do I mean by Parent application and subapplication?

The subapplication is the application model that manages the subcanvas. The parent application is the application model that provides the subapplication as an aspect model.

3.27 What is the `SubCanvasSpec`'s `clientKey`?

`clientKey` is the message name which is sent to parent application to acquire the subapplication that manages the subcanvas.

3.28 What is the `SubCanvasSpec`'s `majorKey`?

`majorKey` is the name of `ApplicationModel` subclass that builds and runs the subcanvas component.

3.29 What is the `SubCanvasSpec`'s `minorKey`?

`minorKey` is name of the class method that returns a full spec describing the subcanvas interface.

3.30 How do I rebuild a subcanvas?

By sending one of the following messages to `aSubCanvas`.

```
client: appModel

client: appModel spec: aSpec

client: appModel spec: aSpec builder: aBuilder
```

`appModel` is subapplication, `aSpec` is full spec of subcanvas, and `aBuilder` is an instance of `UIBuilder`.

3.31 What is `ScheduledControllers`?

`ScheduledControllers` is a global variable that refers to the only instance of `ControlManager`. It is responsible for managing all current window controllers.

3.32 How do I access active window?

```
ScheduledControllers currentController view
```

3.33 How do I access widget of a named component?

```
(anApplicationModel builder componentAt: #compID) widget
```

3.34 How do I give keyboard focus to a widget?

By sending `takeKeyboardFocus` to a VisualWorks component.

3.35 How do I change a component's label?

```
comp labelString: aString
```

3.36 How do I make a widget invisible/visible?

```
comp beInvisible.

comp beVisible.
```

3.37 How do I enable/disable a component?

```
comp enable.

comp disable.
```

3.38 Which method is used by `aSequenceView` to display its contents?

`displayString`

3.39 How do I use an arbitrary method to display contents in `aSequenceView`?

By sending `displayStringSelector: aSymbol` message to `aSequenceView`. `aSymbol` is the name of method used to display strings.

3.40 How do I change grid spacing of `aSequenceView`?

`aSequenceView lineGrid: aNumber`