Weak References

Most object references in VisualWorks are *strong*. If there is a chain of strong references to an object from one of the system roots (e.g., the system dictionary, Smalltalk), then the garbage collector will not reclaim the object. However, if the object is only reachable via one or more chains with at least one *weak* reference in them, then it will be reclaimed. Weak references can only appear (directly) in instances of WeakArray (not even in instances of a subclass).

1. WeakArray

A WeakArray is similar to a plain Array, except that all the indexable instance variables contain weak references. When the garbage collector finds that the last references to an object reside in one or more WeakArrays, it reclaims the object, and sets the instance variables that used to refer to the object to 0. It also initiates a sequence that allows *finalization* to take place (i.e., actions which cleanup after an object has died).

2. Finalization

In addition to zeroing the appropriate element of the WeakArray, the WeakArray is added to an internal VM queue known as the *finalization queue*, and a Semaphore (stored in the FinalizationSemaphore class variable of WeakArray) is sent the message signal. Signalling the Semaphore wakes up the FinalizationProcess (another class variable) that has waited on the Semaphore. This uses another primitive to take the first WeakArray from the finalization queue, and then sends it the message changed: #ElementExpired.

Each WeakArray maintains a list of dependents (in the dependents instance variable), and when it receives changed:, it broadcasts update: to its dependents. It is up to the dependents to take the appropriate finalization action. Note that this occurs *after* the object has died, and hence can no longer be accessed. This means that the finalization has to be planned in advance!

Once a dependent has been informed, it will usually nil out the instance variable of the WeakArray which used to refer to the late object. A special method, forAllDeadIndicesDo:, is provided to assist with this. This iterates over all the indexes that have recently expired, passing each to the block argument. It is written in such a way that it is guaranteed each expiry will only appear once, even in the presence of concurrent activity (see Fig.1).

As an example, consider class Finalizer, defined as a subclass of Object. It has one method, as below:

```
update: anAspect with: aParameter from: aSender
    Transcript cr; show: anAspect printString; tab; show: aSender printString
```

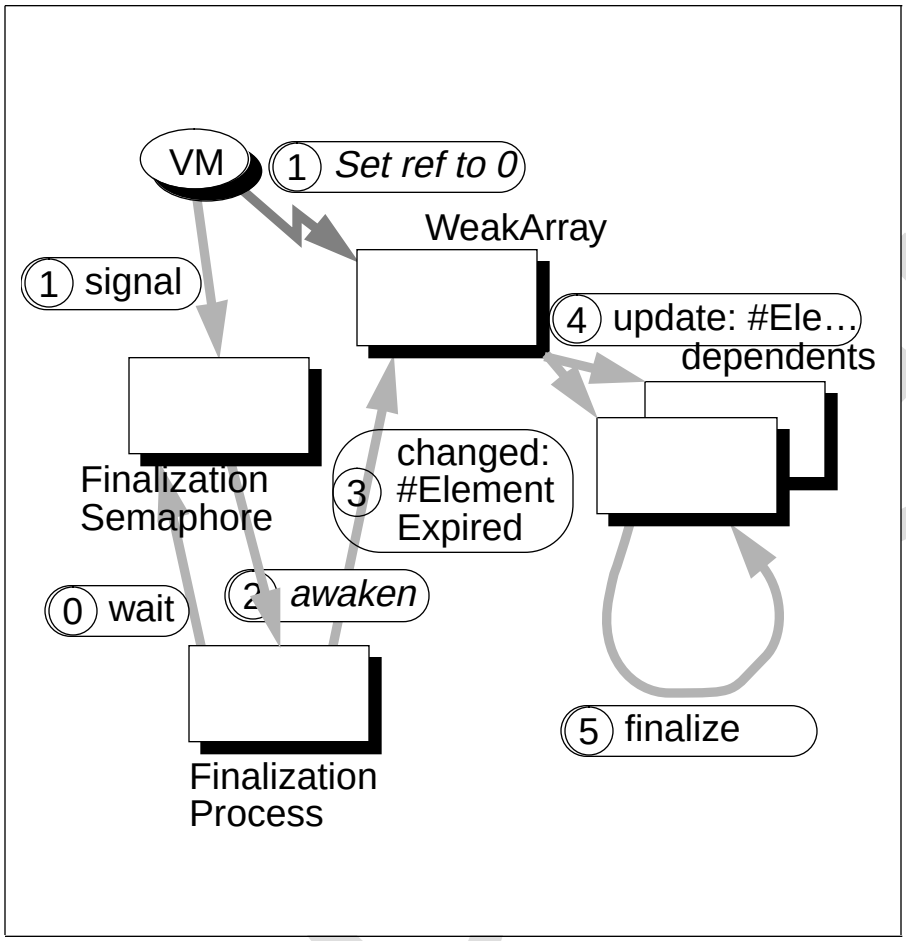

Figure 1: Finalization

The following code creates an instance of Menu, and places it as the sole element of a WeakArray. An instance of Finalizer is then added as a dependent of the WeakArray. The Menu is then sent the message startUp (causing it to appear on the screen). After the user has selected the menu option, the Menu disappears, and an Inspector is opened on the WeakArray.

```
| menu  weakArray |
menu := Menu labels: 'Press me and I die!'.
weakArray := WeakArray with: menu.
weakArray addDependent: Finalizer new.
menu startUp.
weakArray inspect
```

Because there are no other strong references to the Menu, its place in the WeakArray is zeroed out, and the WeakArray is sent a changed: message. The Finalizer, as a dependent of the WeakArray, receives an update:with:from: message, which it responds to according to the method shown above. (Note that in order for this example to work, the WeakArray must outlive the Menu — hence the Inspector.)

Ex 1. Try the example above. Leave the menu on the screen for a variety of different periods of time before making a selection, and observe the delay between the menu disappearing and the message appearing in the Transcript. What's the reason for the variation?

3. WeakDictionary

Common uses of finalization involve a WeakDictionary. This is an IdentityDictionary with weak references for its values. It also maintains a shallow copy of each value to use as an *executor*. When the real value dies, its shallow copy is sent the message finalize.

For example, class FamilyMember is defined below:

```
Object subclass: #FamilyMember
    instanceVariableNames: 'name '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Weak References'
```

It has methods name and name: to access the instance variable. In addition, it has the method finalize, thus:

```
finalize
    Transcript cr; show: name, ' has died'
```

In the code below, an Array is created containing four instances of the class. The Array is used to populate an instance of WeakDictionary, using the names of the FamilyMembers as the keys. When an Inspector is opened on the WeakDictionary, the FamilyMembers no longer have any strong references, hence the shallow copy of each (maintained by the WeakDictionary) is sent the message finalize, causing their name to be written to the Transcript.

```
| array weakDictionary |
array := Array
            with: (FamilyMember new name: 'Jan')
            with: (FamilyMember new name: 'Alex')
            with: (FamilyMember new name: 'Sam')
            with: (FamilyMember new name: 'Hillary').
weakDictionary := WeakDictionary new.
array do: [:f | weakDictionary at: f name put: f].
weakDictionary inspect
```

HandleRegistry is a subclass of WeakDictionary used for recording *handles* on external entities (windows, files, etc.).

Ex 2. Implement class FamilyMember and experiment with the example code.

Ex 3. Browse class WeakArray, WeakDictionary, WeakKeyAssociation and HandleRegistry. Investigate where they are used.