# Return Values

Juanita J. Ewing

Instantiations, Inc.



Derived from Smalltalk Report

Every method has one return value. A return value can be a boolean indicating the success of failure of the operation, a new object that is the result of the operation, or an existing object such as the receiver of the method. The default return value of a method is self, the receiver of the method.

In this column we examine some common return values from methods. Evolution of interfaces can result in an ad hoc variety of return objects that are more difficult for a client to use. A better alternative is specialized objects that encapsulate return values.

**What has return values?**

In Smalltalk there are two structure, composed of a sequence of statements, that have a return value: blocks and methods. These structures both have two kinds of returns, implicit and explicit returns. This column discusses in method returns in detail.

An explicit return consists of a return statement, and causes an immediate return from the method context, even if the return statement is inside a block. The return value is the value of the expression to the right of the return operator (^). The expression ^nameCollection includes: myName returns true or false, depending whether myName is included in nameCollection.

Another form of return is an implicit return. Implicit returns are performed when no explicit return is performed. An implicit return is performed when execution "falls off the end" of the method.

Blocks and methods have different implicit return value semantics. For methods, the implicit return value is the receiver of the method, otherwise known as self. For blocks the return value is the value of the last statement in the block.

**Examples**

Smalltalk class libraries are filled with examples of implicit and explicit returns. The method displayOn: has an implicit return. The implicit return is performed after the last statement in the method.

```
Wedge
displayOn: aGraphicsTool
```

```
    "Graphically display the receiver on <aGraphicsTool>."

    self fillOn: aGraphicsTool
```

The truncated method for Line has an explicit return, but it is the same as the implicit return. Sometimes developers use an explicit return for emphasis, even though it is not necessary. In this case, it is probably because the method, unlike other similar line methods, returns the receiver rather than a copy of the receiver.

*Line*
**truncated**

```
    "Answer the receiver with the coordinates of its end points truncated to integers."

    self start: self start truncated.
    self end: self end truncated.
    ^self
```

Other common return objects are nil, true, false and strings. By convention, a return value of nil usually indicates an exception condition or an error. Many older class libraries written in the days before exception handling had old methods that started out with a return of self, like the method below. Later, some error checking might be added and another return value was used to indicate the error. The other return value was typically nil, which is easy for clients to check for.

**operationFooOn: anObject**

```
    "Perform the operation foo on <anObject>. Return nil if error."

    | fooizedObject |
    (self compatibleWith: anObject)
            ifFalse: [^nil].
    fooizedObject := self prepare: anObject
    self foo: fooizedObject
```

Another common way to indicate an errors is to return a string describing the error. The class method bindTo: checks the return value for the instance method bindTo:. A string indicates an error.

*ObjectLibraryBind class*
**bindTo: aDLLName**

```
"Bind the ObjectLibrary with <aDLLName> into the current image."

| result |
result := self new bindTo: aDLLName.
result isString ifTrue: [ self error: result ].
^result
```

**A Bad Idea**

As operations grow more complex, there is a tendency for the interface to the operation to become broader. Sometimes broadness takes the form of disparate return values, each returned under a different condition. Methods with multiple widely-differing return values require the client to execute conditional code or perform a kind of case statement in order to use the result of the method invocation.

Let's look at an example of a complex operation. The operation has these characteristics:

- It might not succeed.
- The operation has a second chance of success - it can be retried with some input ignored.
- If the operation fails, it might be because of an internal error, or because an external function failed. For debugging purposes, it is desirable to distinguish between the two.
- Another effect of the operation is the creation of an OrderedCollection of strings containing result data from the operation.

One possible solution to the problem of how to return this information is to use different return values to indicate how the operation proceeded. This solution uses the following set of return objects (with their interpretation):

- self — operation completed without error and return object represents success.
- nil — operation completed but some input data was ignored. Return object represents conditional success. If the invocation of the method has user interaction available, it would appropriate here.
- string — operation failed due to internal constraints and return object represents error message.
- integer — operation failed due to external constraints and return object represents error code. Client must look up message in error code table. Desirable to have error code in case error code table is inconsistent with external interface.

This solution also makes use of a global variable to contain the collection of result strings. The operation has a side effect of setting the global variable to the collection.

Clients of this method must test for the kind of return value to interpret the result of the

operation. The testing of the return value and its interpretation looks like a secret code between the method and the client. Occasionally, a developer has to break the secret code to maintain the application. Client code might look this:

invokeOperation

"Invoke the operationWithPoorInterface. Interpret the result and conditionally return the global variable <GlobalResult> if the operation succeeded. If it failed return an empty collection. If the invocation was interactive, notify the user of the operation results."

```
| result errorMessage |
result := self operationWithPoorInterface.
result == self
        ifTrue: [self invocation isInteractive
                ifTrue: [self notifySuccess].
            ^GlobalResult].
result isNil
        ifTrue: [self invocation isInteractive
                ifTrue: [self notifyDataIgnored].
            ^GlobalResult].
result isString
        ifTrue: [self notifyError: result.
            ^OrderedCollection new].
result isInteger
        ifTrue: [errorMessage := (self class errorTable at: result ifAbsent: ['unknown
error'].
            self notifyError: result printString, ': ', errorMessage.
            ^OrderedCollection new]
^self error: 'unable to interpret result of operationWithPoorInterface'
```

The messages that can be sent to the result depend on the type of object. If the return values from a method are highly polymorphic then the client for the method can use the result without testing for the kind of object.

This solution suffers from several problems:

- Ease of Use: the client must map the return value to the correct action, based on the kind of return object. The kinds of return objects can be arbitrary, and can be difficult for the developer to interpret correctly.
- Maintenance: requires the client to have a "case" statement that is error prone during evolution and maintenance - if the set of return values changes then all clients must be updated.

- Encapsulation: a global variable that is set as a side effect of the operation is problematic because it is not protected from modification outside of the operation.

**Another Bad Idea**

The other bad idea is to package multiple return objects into one generic return object, such as an array. We can rework the above example to use an array for a return value. The array contains much of the information above, but in a different form:

- element 1 success boolean. Set to true if the operation succeeded and false if data was ignored or operation failed.
- element 2 data ignored boolean. Set to true if the operation succeeded by ignored some input data, false otherwise.
- element 3 error message. Set to an empty string if the operation succeeded, a descriptive string if it failed.
- element 4 error code. Set to non zero if external operation failed. Error message is also set.
- element 5 collection of result strings. Empty if operation failed.

In this form, we give the operation the responsibility of determining the proper error message, and we do not use a global variable to return the collection of result strings. Now the client must interpret the contents of the array instead of the set of return values. The code for the client might look like this:

```
invokeOperation
    "Invoke the operationWithPoorInterface. Return a collection of strings if the operation
succeeded. If it failed return an empty collection. If the invocation was interactive, notify the
user of the operation results."

    | result errorMessage errorCode |
    result := self operationWithPoorInterface.
    (result at: 1)    "success"
            ifTrue: [self invocation isInteractive ifTrue: [self notifySuccess].
                    ^result at: 5].
    (result at: 2)    "data ignored"
            ifTrue: [self invocation isInteractive ifTrue: [self notifyDataIgnored].
                    ^result at: 5].
    errorMessage := result at: 3.
    errorCode := result at: 4
    errorCode > 0
            ifTrue: [errorMessage :=  (self class errorTable at: errorCode ifAbsent: ['unknown
error']), ': ', errorMessage]
    self notifyError: errorMessage.
```

```
^OrderedCollection new
```

This solution does not have a side effect of setting a global variable. But, there are three reasons why capturing a set of return objects in an array is not a good choice. For a more complete discussion of these issue see the article titled *"Don't Use Arrays?"* in the **Smalltalk Report** vol. 2 no. 7.

- Ease of Use: Clients of the method returning an array need to know arbitrary indices to obtain the data instead of sending messages with meaningful names.
- Encapsulation: Multiple return objects form a coherent set and should have behavior to represent operations on these objects. An array has no way to encapsulate behavior, and consequently, clients of the method need to write much more code to duplicate the behavior of the return set of objects. Each clients writes the same code over and over.
- Information Hiding: With an array representing multiple return objects, the constituent data, including private data, is accessible to all clients. Also, the formation of the objects within the array cannot be changed without affecting all clients.

**A Good Idea**

There is an alternative to packaging different return values in a generic data structure, or returning several kinds of objects. The alternative is a single instance of specialized return object. A specialized return object can encapsulate the success or failure of an operation, errors descriptions, and multiple results. Independent of the success of the operation, all clients can send the same messages to the result. A single return object is easier for clients to use, and encapsulates appropriate behavior.

Using the same problem description, here is a solution that uses a specialized return object. A partial description of it's protocol follows:

SpecializedReturnObject

| | |
|---|---|
| **wasSuccessful** | "Return true if the operation succeeded." |
| **wasDataIgnored** | "Return true if the operation ignored some input data, false otherwise." |
| **errorMessage** | "Return an empty string if the operation succeeded, a descriptive string if it failed. If the failure is external, the message includes the error code." |
| **errorCode** | "Return non-zero if external portion failed, zero otherwise." |
| **stringCollection** | "Return a collection of result strings, empty if the operation failed." |

Using a specialized return object, the client of the operation sends messages to interpret results instead of testing for arbitrary return values or access arbitrary elements in an

array. Client code might look like this:

```
invokeOperation
    "Invoke the operationWithPoorInterface. Return a collection of strings if the operation
    succeeded. If it failed return an empty collection. If the invocation was interactive, notify the
    user of the operation results."

    | result errorMessage |
    result := self operationWithPoorInterface.
    result wasSuccessful
            ifTrue: [self invocation isInteractive ifTrue: [self notifySuccess].
                    ^result stringCollection].
    result wasDataIgnored
            ifTrue: [self invocation isInteractive ifTrue: [self notifyDataIgnored].
                    ^result stringCollection].
    self notifyError: result errorMessage.
    ^OrderedCollection new
```

In this example, special code that dealt with the error code was dropped out. It is not necessary for the client to know the error code for the error message to be composed. Instead, it is composed by the return object. During a debug session, the error code can still be individually accessed.

**A Real Life Example**

Compilation is a complex operation with multiple results. The solution used by Smalltalk/V for compilation results is a specialized return object. an instance of CompilationResult. The behavior for CompilationResult from Smalltalk/V for Win32 follows (simplified for presentation). The behavior can be divided into several categories:

- methods for determining the success of the operation and dealing with errors
- methods for returning the results of successfully compiling some source code
- methods for returning the results of parsing some source code, subdivided into
    - local names
    - selector
    - miscellaneous results, such as the messages sent by the method

*******Note to editor: CompilationResult could be a side bar********

CompilationResult

*success*

**wasSuccessful** "Answer <true> if the receiver represents a successful compilation."

**error** "If the compilation was unsuccessful then return the Compilation Error object that describes the error that terminated compilation."

**errorMessage** "If the compilation was unsuccessful then return a string describing the error that terminated compilation, otherwise return an empty string."

*compilation results*

**association** "Answer an association between the method selector and the compiled method that was created. Answer nil if the compilation was unsuccessful or if what was compiled was not a method"

**method** "Answer the compiled method that was created by the compilation. Answer nil if the compilation was unsuccessful."

*parse results - local names*

**argumentNames** "Answer a list of the names of all method arguments defined within the source."

**localNames** "Answer a list of the names of all variables defined within the source."

**nonLocalNames** "Answer a list of the names of all variables referenced but not defined within the source."

*parse results - selector*

**selector** "Answer the method selector by which the compiled method would normally be invoked. Answer nil if the an expression was compiled or if the compilation was unsuccessful."

*parse results - other*

**messages** "Answer a list of the messages sent within the source."

**primitiveSpecification** "Answer a string containing the primitive specification found in the source code, including the enclosing brackets ('<' and '>'). If the source code had no primitive specification, the answer is an empty string."

**sourceCode** "Answer the source code that was parsed. This may be different from the original source code if the source was modified by an error handler."

**temporaryNames** "Answer a list of the names of all temporary variables and block arguments defined within the source."

Depending on the client, results of compiling could be used to create methods for classes, to build static databases of messages, or to collect data for metric and productivity measurement. Typical use of the interface to the compiler looks like this:

**invokeCompiler**

*"Invoke the compiler. Return a new compiled method if the compilation was successful, nil otherwise."*

```
| compiler compilerResult |
compiler := self compilerClass forClass:  self classToCompileFor .
compilerResult := compiler compile: initialSource.
compilerResult wasSuccessful ifFalse: [ ^nil ].
self install: compilerResult association withSource: compilerResult sourceCode.
self buildCallTableFrom: compilerResult selector to: compilerResult messages.
^self
```

**Advice**

When designing systems with complex operations, pay attention to the interface between the client and the operations. If there is a requirement for multiple return values, consider the use specialized return objects.

Analyze interfaces to existing complex operations. Be wary of

- sets of return objects,
- the use of "case" statements in client code to analyze return values, and
- generic data structures, such as arrays, which are analyzed by the client.

These are signs of places that could benefit from specialized result objects. The results of rework with specialized objects will more understandable, extensible, maintainable and reusable.