

Discussion Paper on Wrapper Class Constructors in Objective Modula-2

Document version: 1.02 -- Date: 2005-08-20 -- Author: BK

1) Introduction

This paper discusses how non-object data structures can be elevated to first-class status by providing a facility to automatically wrap record types into container classes.

Assuming a universal stack storage class Stack ...

```
INTERFACE MODULE Stack : NSObject;

// return a new initialised stack object with initial capacity cells
CLASS METHOD initWithCapacity: (cells : CARDINAL) : OBJECT;

// return true if the receiver is empty, otherwise false
INSTANCE METHOD isEmpty : BOOLEAN;

// place object obj on top of the receiver
INSTANCE METHOD push: (obj: OBJECT);

// return the receiver's topmost object
INSTANCE METHOD pop : OBJECT;

END Stack.
```

... which allows to place any kind of object on an object of class Stack regardless of the type of object. It even allows to mix objects of different types. Yet, the stack object cannot accommodate any variables which are not objects.

In order to place non-object variables on a stack object, the following two approaches exist:

- 1) define a non-universal stack class, for example a stack of INTEGERS
- 2) define a wrapper class to hold the non-object variables as instance variables

Approach #1 has the disadvantage that the benefits of dynamic typing are lost. It is necessary to create one class each for every type of variable and that class can only accept one type of variable.

Approach #2 does not give up the benefit of dynamic typing, but it is somewhat inconvenient to write the wrappers if the sole reason for creating the wrapper class is to have an object which can hold variables which are not objects.

Based on this observation, it may be beneficial to provide additional syntax and semantics so as to take the inconvenience out of approach #2. How to design such a facility is the subject of this paper.

2) Wrapper Classes with Public Instance Variables

Assuming an existing record type ...

```
TYPE
  Colour = RECORD
    red, green, blue : CARDINAL [0..255];
  END;
```

... in a situation where a variable of type Colour needs to be placed on the universal stack storage

provided by the Stack class described further above.

A wrapper class to hold variables of type Colour will be required. In order to ease the creation of such a wrapper class the following syntax could be provided ...

```
TYPE
    ColourClass = PUBLIC CLASS OF Colour;

... from which the compiler would then construct a wrapper class as equivalent to the following ...

INTERFACE MODULE ColourClass : NSObject;

INSTANCE VAR
    PUBLIC TO ALL
    red, green, blue : CARDINAL [0..255];

// no methods will be declared for this class
END ColourClass.
```

At this point, the class could be used as follows ...

```
MODULE Colours;

FROM SomeLib IMPORT Colour;

IMPORT ColourClass, Stack;

VAR
    fooColour : Colour = { 50, 100, 200 };
    barColour : ColourClass = [[ColourClass alloc] init]
    stack : Stack = [Stack initWithCapacity:20];

BEGIN
    barColour^ := fooColour;
    [stack push:barColour];
END Colours.
```

... which still requires an explicit intermediary step of copying fooColour to barColour before it can be put on the stack. This is an improvement, but there is still room for further improvement.

One improvement would be to extend structured value constructors to wrapper classes ...

```
barColour := ColourClass { 50, 100, 200 };
```

... which would be equivalent to ...

```
barColour^.red := 50; barColour^.green := 100; barColour^.blue := 200;
```

... but it would be even more convenient to allow the following ...

```
[stack push:ColourClass { 50, 100, 200 }];
```

Objective-C already has something similar for strings ...

```
s = [NSString stringWithString:
    @"String object created on the fly from a string literal"];
```

where @"some string" can be used anywhere a const *NSString expression is allowed.

Consequently, the extension of structured value constructors of type RECORD to wrapper classes would merely make use of an existing concept already present in the language. The same limitations would apply, most notably that the object implicitly created from the literal is treated as a constant to which there is no write access.

3) Wrapper Classes with Private Instance Variables

A similar facility could be provided either in addition to the above described wrapper class or as an alternative. Instead of creating wrapper classes with public instance variables and no accessors and mutators, this facility would create wrapper classes with private instance variables, accessors and mutators ...

```
TYPE
    ColourClass = PRIVATE CLASS OF Colour;
```

... from which the compiler would then construct a wrapper class as equivalent to the following ...

```
INTERFACE MODULE ColourClass : NSObject;

FROM SomeLib IMPORT Colour;

INSTANCE VAR
    PRIVATE
        red, green, blue : CARDINAL [0..255];

CLASS METHOD newWithColour: (colour : Colour) : OBJECT;

INSTANCE METHOD red : CARDINAL [0..255];

INSTANCE METHOD setRed: (value : CARDINAL [0..255]);

INSTANCE METHOD green : CARDINAL [0..255];

INSTANCE METHOD setGreen: (value : CARDINAL [0..255]);

INSTANCE METHOD blue : CARDINAL [0..255];

INSTANCE METHOD setBlue: (value : CARDINAL [0..255]);

INSTANCE METHOD colour : Colour;

INSTANCE METHOD setColour: (colour : Colour);

END ColourClass.
```

... which would have more overhead than the public wrapper class but follow the OO paradigm, especially the concept of information hiding. It also allows compliance with KVC/KVO rules.

4) Applicable Rules

If the facility was to be introduced, then the following rules should apply:

- 1) the identifier of fields of the record being wrapped are considered to start with a lowercase character, even if they don't. A field with an identifier "Red" would create a corresponding instance variable "red", also a corresponding accessor "red" in case a private wrapper is created.
- 2) The structured value constructor should be limited to wrapper classes and only include instance variables, not any IMPs (pointers to method implementations).
- 3) The construction of wrapper classes should be limited to named records. The use of inline records should be explicitly forbidden. Thus ...

```
TYPE
    FooBarClass = CLASS OF RECORD foo : Foo; bar: Bar; END;
```

... should generate a compile time error because it doesn't provide a reference to a named record type.

This is not strictly necessary but it should be applied for reasons of language hygiene. That is to say, the facility is considered to be a wrapper for existing record structures and nothing else. In order to create a class from scratch, the class module syntax should be used.

This facility could in principle be applied to types other than RECORD, for example ARRAYS. Yet, again, for reasons of language hygiene, it should only be provided for use with RECORD types. RECORD is most "compatible" with the OO paradigm, and it makes it easy to follow KVC/KVO rules.

END OF DOCUMENT