# Evolution of the Constants

## Or 'how to build your own enumerated data type'

WRITTEN BY
ROLAND MÜHLBERGER

**PowerBuilder comes with a wide range of enumerated data types, and the values are easily distinguished by the exclamation mark at the end. However, there is no way to build your own enumerated data type in PowerScript, and it seems that even PB 10 won't help us that much here. (Keep your fingers crossed that user-defined enumerateds will be a part of PB 11.) This article shows how to build a self-describing and robust enumerated data type and discusses the ideas behind it.**

### Stage 0: Using Numbers

From the beginning PowerBuilder programmers used numbers to code states in their programs. For example, think about a ticket system: each ticket has a priority. The intrepid programmer thought the following: "Let's use numbers for priorities. This way I can work with the numbers, store them in longs, compare them, and store them in the database as well." "1" was used for low, "2" for medium, "3" for high. This led to code like the following, assuming we have a class "Ticket" that has an instance variable il_prio holding the priority:

```
function Boolean of_IsHighPriority()
// 3 means high prio
RETURN (this.il_prio = 3)
```

### Stage 1: Local Constants

Now we move one step forward: PowerBuilder can do constants, can't it? If we move the 3 to a constant, the code will be much more readable and doesn't need the comment anymore:

```
function Boolean of_IsHighPriority()
constant Long HIGH = 3
RETURN (this.il_prio = HIGH)
```

This is a big leap forward. But what if we need to use the priority values outside this function? This problem leads to the following.

### Stage 2: Constants as Instance Variables

We want to use constants throughout our class, so we simply move it to the instance variables section. In it we find code similar to the following:

```
constant Long LOW = 1
constant Long MEDIUM = 2
constant Long HIGH = 3
```

Our function can now be changed to:

```
function Boolean of_IsHighPriority()
RETURN (this.il_prio = HIGH)
```

Sometimes this kind of usage is sufficient, but we want to use the constant outside our own class. Easy, you say, simply make the instance variables public! This leads us to Stage 3.

### Stage 3: Public Constant Instance Variables

Make your constants public (which they already are by default if you don't state differently). To make it clear, here they are again:

```
public:
constant Long LOW    = 1
constant Long MEDIUM = 2
constant Long HIGH   = 3
```

Internally we can simply use the constant, but how do we use it from outside? Of course, you can use an object reference and access its public attributes like this:

```
Ticket n_ticket

n_ticket = CREATE Ticket

n_ticket.of_SetPrio(n_ticket.HIGH)
```

There is another way to use the constant. Did you know that PowerBuilder declares a global variable of each and every class you implement, be it a window, a nonvisual object, or anything else? Of course you did. The name of the global variable is exactly the same as the name of the class. You can prove this by looking at the exported sources, where you'll find a row similar to:

```
n_myclass n_myclass
```

at the beginning of the script (for an arbitrary class n_myClass).

We can use this global class, and we can even use its constant instance variables without bothering whether the class gets instantiated or not. For our class Ticket, we can write the following:

```
Ticket n_ticket

n_ticket = CREATE Ticket

n_ticket.of_SetPrio(Ticket.HIGH)
```

We never instantiated the global reference "ticket," but the code still compiles and runs okay. It works because PowerBuilder replaces constant variables with their actual values at compile time (compiling into pcode, of course). This explains the fact that if you change the value of a constant variable, you first need to regenerate all classes that use it, otherwise they will still hold the previous wrong value.

Back to our example. If we read the sources, something is not quite right. Tickets are not high or low, but priorities are. We need to advance to Stage 4.

### Stage 4: Constant Classes

We simply move the constants out of the Ticket class into their own class, Priority, that holds the values above.

The class Priority doesn't need to get instantiated; we can use it like this:

**AUTHOR BIO**

Roland Mühlberger works as a PowerBuilder class librarian and software engineer for the Austrian company ecosys. In addition, he runs his owns business (ROMU Software) as an independent consultant. His special interest (besides mountain climbing) are programming tools; he's the author of SmartPaste, a tool for documenting PB source code.

```
function Boolean of_IsHighPriority()
RETURN (this.il_prio = Priority.HIGH)
```

That seems to be very versatile and self-explanatory as well. In fact, this is as far as many PowerBuilder shops have come. But let me show you some drawbacks of this system.

Suppose you have the function to set the ticket priority of_SetPriority:

```
subroutine of_SetPriority(Long
al_prio)
```

Nobody can ensure that somebody calls this function in this way:

```
n_ticket.of_SetPriority(1000)
```

Because of this, somebody thinks that 1,000 is a reasonably high number for a really high priority, so we need to check the values inside of_SetPriority. There may be some more places where you have an input of type "Priority", and all these places need the according checks:

```
IF (al_input <> Priority.Low) OR &
   (al_input <> Priority.Medium) OR &
   (al_input <> Priority.High) THEN
   // wrong input
END IF
```

The check is okay, but this is not really what we want. We can move the check off to a function, but we still need to do the checks every time we write a function that gets a priority as input.

The next step is a huge leap forward.

## Stage 5: Encapsulating the State

In our class Priority, we add an instance variable of type ULong (so we don't have to deal with negative values, we will need that later on):

```
private Ulong iul_value
```

Together with this we write accessor methods: of_GetValue and of_SetValue. In of_SetValue we implement the range check so no one can send us wrong values:

```
function ULong of_GetValue
RETURN iul_value

function Integer of_SetValue(Long
aul_value)
IF (aul_value <> Low) OR &
   (aul_value <> Medium) OR &
   (aul_value <> High) THEN
// wrong input
   RETURN -1
ELSE
   iul_value = aul_value
   RETURN 1
END IF
```

In order not to deal with creating and destroying the class each time we use it, we make the class Priority autoinstantiate. This way we can easily use the constants of the classes as well as an encapsulated value. But we need to change the interface of the of_SetPriority method of the Ticket class to accept Priority instead of Long values. Also the ticket class does not have an instance of type Long, but an instance of type Priority:

```
private:
Priority ie_prio
…
subroutine of_SetPriority(Priority
ae_prio)
ie_prio = ae_prio
```

Together, with the check above, our code is now pretty robust. There is no way to set the ticket priority to a wrong value (supposing we initialize the Priority to Low in its Constructor). Code written in the way it is above is also much more self-describing: the interface of the method does not include a Long, but a very strict Priority data type. This is already a simple, self-built enumerated data type.

## Stage 6: Enhanced Version Using Metadata

We can enhance the previous example by using metainformation about classes in order to build the list of valid values. If we know the constants of the class, we can build a list of possible values. To get the list of constants, we use PowerBuilder's system: ClassDefinition and VariableDefinition (see Listing 1). We fetch the ClassDefinition of our constant class by referencing the attribute classDefinition, then we loop through all variables of the class and check whether they're constants or not. If they are and if they are ULongs, they're added to an array. The array holds strings, and the strings are the names of the constants. We use those names later on to do a conversion between the numeric and the string representation of the enumerated value. As you can see in the listing, the array called is_valueNames is at the instance level.

Now you know why we needed unsigned longs as data types for our constants: we use the constants as an index into the array.

We put the above script in a function called of_InitValueNames. To make the function available for all our constant classes, we create a base class for our enumerateds called Enum. of_InitValue Names is called in the constructor of the class.

Now that we know all possible values, we can implement a generic version of the of_SetValue function. This function checks whether there is a name set in the array at the index we want the value set to. If there is, we know that one of our constants is called that way and therefore exists.

```
function Integer of_SetValue(Long
aul_value)
i_ret = -1
IF aul_value > 0 AND NOT
IsNull(aul_value) THEN
   IF UpperBound(is_valueNames[]) >=
aul_value THEN
      IF is_valueNames[aul_value] <>
'' THEN
         iul_value = aul_value
         i_ret = 1
      END IF
   END IF
END IF
```

Now that we know the names of the possible values, we can even allow the use of strings to set and get the values. Therefore we can implement two more functions:

```
of_SetValue(String) and of_GetName()
```

of_SetValue can be used to set the

> "Another way of optimizing is to shorten the syntax for using the enumerated"

internal value using one of the names of the constants as strings; of_GetName yields the name of the internal value as string. In these functions, we ignore the case and always send the name in lower case.

As an enhancement to this, we implement a function called of_GetValidNames(ref String []) that can be used to get all valid names for the internal value. (The list of names could be used to populate dropdown listboxes.)

Let's sum up what the current state is: we have an "enumerated base class" Enum that incorporates functions for setting and getting the encapsulated value. The value can be set by using numbers or its name. To use the system, we need to inherit from this class and implement the possible values as constant instance variables holding values starting from 1.

Our class Priority is therefore now derived from Enum and the only things to be scripted are the instance constants already mentioned. This is true for any enumerated data type you want to implement: simply derive from Enum and implement constant instance variables.

To use a value of our Priority enumerated, we now need to do the following:

```
Priority p

p.of_SetValue(Priority.High)
obj.of_SetPriority(p)
```

**Listing 1**

```
Long    l_cur, l_count
String s_varName
ULong   ul_initVal

ClassDefinition cd_myClass
VariableDefinition vd_var

cd_myClass =
this.classDefinition

l_count =
UpperBound(cd_myClass.variableL
ist)
FOR l_cur = 1 TO l_count
 vd_var =
cd_myClass.variableList[l_cur]
 IF vd_var.isConstant THEN
  IF
ClassName(vd_var.initialValue)
= 'unsignedlong' THEN
   ul_initVal =
vd_var.initialValue
   s_varName = vd_var.name
   is_valueNames[ul_initVal] =
s_varName
  END IF
 END IF
NEXT
```

DOWNLOAD THE CODE!
www.sys-con.com/pbdj/

## Stage 7: Optimization

A few things come to mind when looking at the class right now: it is autoinstantiate, so the list of names will be constructed every time the constant class is used in a script. To overcome this problem, we postpone the construction of the list until it's really needed (see the final source at www.romu.com/download).

Another way of optimizing is to shorten the syntax for using the enumerated. We can do this by using an undocumented feature of PowerBuilder called "indirect variables." Indirect variables are instance variables that don't exist at runtime, but access to them is forwarded to a getter and setter method. The syntax for using indirect variables:

```
public:
indirect ulong val
{of_SetValue(*value),of_GetValue()}
```

If we add such a declaration to the instance variables of the Enum base class, we can use our enumerateds this way:

```
Priority p

p.val = Priority.High
obj.of_SetPriority(p)
```

Setting p.val calls of_SetValue and using p.val in expressions ('reading the value') calls of_GetValue.

As this is an undocumented feature, it might be removed without warning in a future version of PowerBuilder. But I think this is the way PB works: when we access instance variables of controls (for instance, the text of the control), not only the internal representation is changed but also the text on the screen, so we should be quite safe to rely on it. On the other hand, who knows for sure.

Another optimization we can make is to use exception handling if somebody tries to set a wrong value instead of yielding a negative return value or calling a MessageBox. In this case, we use the unchecked version of the PowerBuilder exception system: RunTimeError. This class allows us to throw exceptions that the caller does not explicitly need to catch or declare in its method interface.

## The Resulting Class

The resulting class Enum has the following instance variables:

```
private:

// metadata:
// array holding names of enum con-
stants
// index into array is enum value
String    is_valueNames[]
// already initialized
Boolean   ib_init

// actual data:
// current value
ULong     iul_value

// indirect access
public:
indirect ULong val
{of_SetValue(*value),of_GetValue()}
```

and the following functions:

```
private subroutine of_initvaluenames
()
public subroutine of_getvalidnames
(ref string as_names[])
public function string of_getname ()
public function unsignedlong of_get-
value ()
public function unsignedlong of_get-
valueforname (readonly string
as_value)
public function string of_getnamefor-
value (unsignedlong aul_value)
public function integer of_setvalue
(readonly string as_value)
public function integer of_setvalue
(unsignedlong aul_value)
```

Only of_GetValue and of_SetValue (or .val) need to be used to set and get a value; the other functions can be seen as class methods that yield a description about the data type.

## Summary

By encapsulating a value in a class and using metadata functions of PB, we built a robust, self-describing version of a user-defined enumerated data type that can even be used as a string incarnation. To build your own enumeration, simply inherit from the base class Enum and script the possible values as constant instance variables. Values can be used (and assigned to each other) easily because of their autoinstantiation. A wrong assignment to enumerateds leads to an exception.

The class Enum, the Class Priority, a window showing usage, and more examples can be downloaded from www.romu.com/download. ▼

office@romu.com