

An array of Class Types using *Class Of*

One nifty trick I like to use when utilising *abstract factories*, is to create an array of class types based on a root or parent class.

Consider a class structure something like this

Type

```
ParentClass = class
End;

Child1 = class(ParentClass)
End;

Child2 = class(ParentClass)
End;

Child3 = class(ParentClass)
End;
```

Using the following syntax, I create an array of Classes based on the ParentClass;

Type

```
ClassOfParent = class of ParentClass;
```

Const

```
nChildren = 3;
ChildClassArray : Array[1..nChildren] of ClassOfParent =
    (Child1, Child2, Child3);
```

Now, I have an array that at runtime I can iterate through all based on the ParentClass. Just utilising the classname (*you would probably have a more meaningful class function to query*) it is now a simple matter of iterating the ChildClassArray to find and instantiate the respective child.

```
Function CreateChild(ClassName : String) : ParentClass;
```

Var

```
    nLoop : Integer;
```

Begin

```
    Result:=Nil;
```

```
    For nLoop:=1 to nChildren do
```

```
        If ChildClassArray[nLoop].ClassName = ClassName then
```

```
            Begin
```

```
                Result:=ChildClassArray[nLoop].Create;
```

```
            Break;
```

```
            End;
```

```
End;
```

A (hopefully) more meaningful example:

I need to create a particular document based on a customer type. If the customer is a business customer I must differentiate between customers with a credit limit of 100K or less, and 500K or less. There is no provision for business customers with a credit limit greater than 500K, so the calling code would need to check for a nil result from the CreateDocument function.

Type

```

TCustomer = class
  CustType      : String;
  CreditLimitK  : Integer;
end;

TDocument = class
  class function Supports(Customer : TCustomer) : Boolean;virtual;abstract;
end;

TPrivateCustomerDocument = class(TDocument)
  class function Supports(Customer : TCustomer) : Boolean;override;
end;

TBusinessCustomerDocument100K = class(TDocument)
  class function Supports(Customer : TCustomer) : Boolean;override;
end;

TBusinessCustomerDocument500K = class(TDocument)
  class function Supports(Customer : TCustomer) : Boolean;override;
end;

TGovernmentCustomerDocument = class(TDocument)
  class function Supports(Customer : TCustomer) : Boolean;override;
end;

```

Now instead of making the decision in our calling code (if CustType = etc), we can use the class array concept.

Type

```
ClassOfTDocument = class of TDocument;
```

Const

```

nDocuments = 4;
DocumentsArray : Array[1..nDocuments] of ClassOfTDocument =
  (TPrivateCustomerDocument, TBusinessCustomerDocument100K,
   TBusinessCustomerDocument500K, TGovernmentCustomerDocument);

```

The **CreateDocument** function iterates through our documents array, and queries the class function **Supports**. The first child document that supports the customer will be created.

```

Function CreateDocument(Customer : TCustomer) : TDocument;
Var
  nLoop : Integer;
Begin
  Result:=Nil;
  For nLoop:=1 to nDocuments do
    if DocumentsArray[nLoop].Supports(Customer) then
      Begin
        Result:=DocumentsArray[nLoop].Create;
        Break;
      End;
  End;
End;

```

The Supports class functions could be implemented something like this:

```
{ TPrivateCustomerDocument }
class function TPrivateCustomerDocument.Supports(Customer : TCustomer): Boolean;
begin
  Result:=Customer.CustType = 'PC';
end;

{ TbusinessCustomerDocument100K }
class function TBusinessCustomerDocument100K.Supports(Customer : TCustomer): Boolean;
begin
  Result:=(Customer.CustType = 'BC') and (Customer.CreditLimitK <= 100);
end;

{ TbusinessCustomerDocument500K }
class function TBusinessCustomerDocument500K.Supports(Customer: TCustomer): Boolean;
begin
  Result:=(Customer.CustType = 'BC') and
    (Customer.CreditLimitK > 100) and
    (Customer.CreditLimitK <= 500);
end;

{ TGovernmentCustomerDocument }
class function TGovernmentCustomerDocument.Supports(Customer : TCustomer): Boolean;
begin
  Result:=Customer.CustType = 'GC';
end;
```

... and finally some calling code:

```
aDocument:=CreateDocument(Customer);
if Assigned(aDocument) then
```

From a software-engineering viewpoint – I like this approach as it greatly simplifies the calling code, and makes it really easy to add new document types without actually writing any new code (beyond the new class itself).

For instance, say we implement a **TBusinessCustomerDocumentNoLimit** class, and need to plug it into our architecture.

Type

```
ClassOfTDocument = class of TDocument;
```

Const

```
nDocuments = 5;
DocumentsArray : Array[1..nDocuments] of ClassOfTDocument =
  (TPrivateCustomerDocument, TBusinessCustomerDocument100K,
   TBusinessCustomerDocument500K, TgovernmentCustomerDocument,
   TBusinessCustomerDocumentNoLimit);
```

```
{ TbusinessCustomerDocumentNoLimit }
class function TBusinessCustomerDocumentNoLimit.Supports(Customer: TCustomer):
Boolean;
begin
  Result:=(Customer.CustType = 'BC') and (Customer.CreditLimitK > 500);
end;
```

* * * * *

In the next 5 minute article, we'll return to this, and demonstrate a nifty technique were we don't need the equivalent of the CreateChild or CreateDocument functions, but from the parent class's *constructor* itself.