Singletons in Delphi

1 Was ist überhaupt ein Singleton?

Ein Singleton in der abstrakten Informatik ist ein Entwurfsmuster das festlegt, dass für eine Klasse maximal nur eine Instanz erzeugt werden kann. Klingt kompliziert?
Okay ein Beispiel:
Wir erstellen uns eine Klasse *TBundeskanzler*. Da es höchstens einen Bundeskanzler geben kann muss unsere Klasse *TBundeskanzler* als Singleton definiert werden. Damit vermeiden wir wie gesagt, dass in einem Programm mehrere Bundeskanzler erzeugt werden können. Wenn wir nun in unserem Programm auf den TBundeskanzler zugreifen wollen erhalten wir jedes Mal dieselbe Instanz vom TBundeskanzler.

2 Das alles ist kein Hexenwerk

Wir fangen damit an unsere Klasse zu definieren:

```
type
	TBundeskanzer = class
		private
			m_sName:String;
			m_nAlter:Integer;
end;
```

Was uns jetzt noch fehlt ist ein *Construktor*, ein *Destructor* und eine Funktion die wir *getInstance* nennen. Letztere wird uns unsere eine Instanz von der Klasse *TBundeskanzler* zurückliefern. Damit nicht jeder beliebig viele Instanzen von *TBundeskanzler* erstellen kann, müssen wir zunächst den *Constructor* - anders als sonst - als private deklarieren, d.h. nur aus der Klasse *TBundeskanzler* heraus kann auf den *Constructor* zugegriffen werden.
Wir erweitern unsere Klasse wie folgt:

```
type
	TBundeskanzler = class
		private
			m_sName:String;
			m_nAlter:Integer;
			constructor create;
		public
			destructor destroy; override;
			class function getInstance:TBundeskanzler;
end;
```

3 Class function

Was jetzt beim genaueren Hinsehen auffällt ist der Begriff *class function*. Manche sind bestimmt schon mal drüber gestolpert manche aber auch nicht. Diese Methode steht nur Klassenreferenzen zur Verfügung, keinen Instanzreferenzen. Hier ein Beispiel:

```
  var BK:TBundeskanzer;
begin
  // BK muss natürlich zuvor noch initialisiert werden
  TBundeskanzer.getInstance; ← Funktion steht zur Verfügung
  BK.getInstance;            ← Funktion wird nicht gefunden
end;
```

Was uns jetzt noch fehlt ist ein Objekt auf das wir unsere Klasse beziehen können, also definieren wir global im Interfaceteil unseres Codes:

```
var g_BK:TBundeskanzer;
```

4 Constructor, Destructor und getInstance

Nun gilt es nur noch unsere 3 Funktionen zu füllen! Wir tun das im *Implementationsteil* wie folgt:

```
Constructor TBundeskanzler.create;
begin
        Inherited create;
end;

Destructor TBundeskanzler.destroy;
begin
        if g_BK = self then
                g_BK := nil;
        inherited destroy;
end;

Function TBundeskanzer.getInstance:TBundeskanzler;
begin
        if g_BK = nil then
                g_BK := TBundeskanzler.create;
        result := g_BK;
end;
```

Das war’s auch schon! Was aber passiert hier nun genau?
Wenn wir *TBundeskanzer.getInstance* aufrufen, wird das Objekt *g_BK* (unserer globaler Bundeskanzer [hoffentlich wird es so etwas niemals geben… ;-)]) initialisiert falls er noch nicht initialisiert wurde. Dazu wird der nach außen hin versteckte *Constructor* der Klasse aufgerufen. Sollte das Objekt allerdings schon initialisiert worden sein, so wird der *g_BK* zurückgeliefert.
Wird nun der Destructor aufgerufen so wird erst dafür gesorgt, dass das Objekt *g_BK* auf nil zeigt und dann wird das eigene Objekt zerstört.
Somit haben wir einen einfachen Weg gefunden zu vermeiden, dass mehrere Instanzen einer Klasse erstellt werden! War doch gar nicht so schwer oder? ☺

5 Der gesamte Quelltext

```
unit Bundeskanzler;

interface

type
	TBundeskanzler = class
		private
			m_sName:String;
			m_nAlter:Integer;
			constructor create;
		public
			destructor destroy; override;
			class function getInstance:TBundeskanzler;
end;

var g_BK:Tbundeskanzler;

implementation

Constructor TBundeskanzler.create;
begin
	Inherited create;
end;

Destructor TBundeskanzler.destroy;
begin
	if g_BK = self then
		g_BK := nil;
	inherited destroy;
end;

Function TBundeskanzer.getInstance:TBundeskanzler;
begin
	if g_BK = nil then
		g_BK := TBundeskanzler.create;
	result := g_BK;
end;
```